

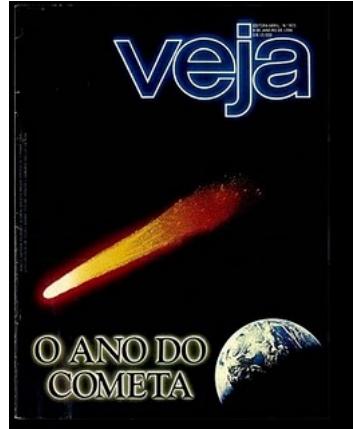
Universidade Federal de Goiás – UFG
Instituto de Informática – INF
Bacharelados (Núcleo Básico Comum)

Algoritmos e Estruturas de Dados 1 – 2020/2

Lista de Exercícios nº 01 – Revisão de Introdução à Programação
(Turmas: INF0061/INF0286 – Prof. Wanderley de Souza Alencar)

Sumário

1	cometa (+)	2
2	José (+)	4
3	overflow (++)	6
4	capicua (++)	8
5	Computação (++)	10
6	Envelopes (++)	12
7	sapos (+++)	15
8	primos (++)	18
9	vetores (+++)	20
10	cálculo de áreas (+++)	22
11	manipulação de matrizes 1 (++)	24
12	manipulação de matrizes 2 (++++)	27
13	Números de Fibonacci (++)	29
14	Fatoração de Números de Fibonacci (++++)	31



1 cometa (+)



(+)

O cometa Halley é um dos cometas de menor período do Sistema Solar, completando uma volta em torno do Sol a cada 76 anos. Na última ocasião em que ele se tornou visível do planeta Terra, em 1986, várias agências espaciais enviaram sondas para coletar amostras de sua cauda e assim confirmar teorias sobre sua composição química. Saiba mais sobre ele em <http://astro.if.ufrgs.br/solar/halley.htm>.

Escreva um programa C que, dado o ano atual, determina qual o próximo ano em que o cometa Halley será visível novamente no planeta Terra. Se o ano atual é um ano de passagem do cometa, considere que o cometa já passou nesse ano, ou seja, considere sempre o próximo ano de passagem após o atual.

Observação: Não se esqueça de considerar os anos bissextos, ou seja, que a cada quatro anos (em direção ao futuro ou ao passado) há um *erro* de um dia em relação ao ano solar que, neste caso, é considerado como tendo exatamente 365 dias terrestres. O ano de 1986, quando o cometa de Halley se tornou visível na Terra pela última vez, é considerado o “*marco de sincronismo*” para os cálculos do programa a ser elaborado.

Entrada

A única linha da entrada do programa contém um único inteiro A , indicando o ano atual, sendo que $0 \leq A \leq 10^4$.

Saída

Seu programa deve imprimir uma única linha, contendo um número inteiro, indicando o próximo ano em que o cometa Halley será visível novamente do planeta Terra.

Exemplos

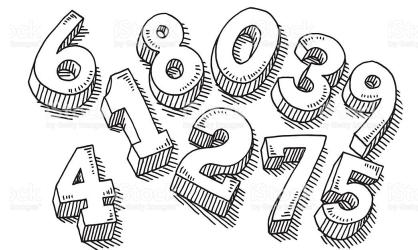
Entrada	Saída
2635	2670

Entrada	Saída
2010	2062

Entrada	Saída
2270	2290

Entrada	Saída
1910	1986

Entrada	Saída
460	465



2 José (+)



(+)

João tem um irmão mais novo, José, que começou a ir à escola e já está tendo problemas com números. Para ajudá-lo a “*pegar o jeito*” com a escala numérica, a professora de José escreve dois números de três dígitos e pede ele para comparar esses números.

Entretanto, ao invés de interpretá-los com o dígito mais significativo à esquerda, ele deve interpretá-los com o dígito mais significativo à direita. Ele tem que dizer à professora qual o maior dos dois números.

Escreva um programa em C que seja capaz de verificar as respostas de José.

Entrada

A entrada conterá uma única linha com dois números de três dígitos, A e B, os quais não serão iguais.

Saída

A saída deve conter, uma linha com o maior dos números na entrada, comparados conforme descrito no enunciado da tarefa. O número deve ser escrito invertido, para mostrar a José como ele deve lê-lo e, havendo zeros à esquerda do número, eles não devem ser escritos, como mostra o quatro exemplo a seguir.

Exemplos

Entrada	Saída
483 583	385

Entrada	Saída
493 583	394

Entrada	Saída
493 589	985

Entrada	Saída
160 720	61

Entrada	Saída
100 200	2



3 overflow (++)



Os computadores digitais foram inventados para realizar cálculos muito rapidamente e, atualmente, atendem a esse requisito de maneira extraordinária. Porém, nem toda “*conta*” pode ser feita num computador digital típico, pois ele não consegue representar todos os números dentro de um único endereço de memória.

Num típico, e simples, computador pessoal atual, por exemplo, o maior inteiro que é possível representar numa unidade de sua memória é $18.446.744.070.000.000.000$ ($2^{64} - 1$). Caso alguma “*conta*” executada pelo computador dê um resultado acima desse número, ocorrerá o que chamamos de *overflow*, que é quando o computador faz uma “*conta*” e o resultado não pode ser representado por ser maior do que o valor máximo permitido (em inglês *overflow* significa *trasbordar*).

Por exemplo, se um computador fictício somente pode representar números menores ou iguais a 1023 ($2^{10} - 1$) e mandarmos ele executar a conta $1022 + 5$, vai ocorrer um *overflow*, já que o resultado deste cálculo é maior que 1023.

Elabore um programa C que seja capaz de receber o maior número que um computador consegue representar em sua memória e uma expressão de soma ou de multiplicação entre dois inteiros positivos, determine se ocorrerá, ou não, *overflow* naquele computador.

Entrada

A primeira linha da entrada contém um inteiro N representando o maior número que o computador consegue representar.

A segunda linha contém um inteiro N_1 , seguido de um espaço em branco, de um caractere C (que pode ser ‘+’ ou ‘x’, representando os operadores de *adição* e de *multiplicação*, respectivamente), de outro espaço em branco, e, finalmente, de outro número inteiro N_2 .

Assim, a segunda linha da entrada representa a expressão $N_1 + N_2$, se o caractere C for ‘+’, ou $N_1 \times N_2$, se o caractere C for ‘x’.

Saída

Seu programa deve imprimir a palavra ‘*overflow*’ se o resultado da expressão causar um *overflow* no computador, ou a expressão ‘no *overflow*’ caso contrário.

Ambas as palavras devem ser escritas com todas as letras minúsculas.

Exemplos

Entrada	Saída
57 20 x 3	overflow

Entrada	Saída
10 5 + 6	overflow

Entrada	Saída
10 5 + 4	no overflow

Entrada	Saída
57 12 x 3	no overflow

Entrada	Saída
30 4 x 4	no overflow



4 capicua (++)



(++)

O pequeno estudante Alan Mathison Turing está aprendendo a decompor um número em unidades, dezenas, centenas, unidade de milhar, dezena de milhar, etc. Ele está com grandes dificuldade neste processo. Sua professora, Ada Lovelace, preocupada com o rendimento de Alan decidiu ensiná-lo por meio de uma brincadeira:

Alan deve pegar um número com quatro algarismos e verificar se o reverso deste número é ele próprio. Se for, Alan deve responder *yes*, do contrário deve responder *no* – Alan é britânico e, por isso, responde em inglês.

Em verdade, Ada está ensinando quando um número é chamado de *capicua* (ou também conhecido por *palíndromo*), pois um número é dito *capicua* quando seu reverso é ele próprio.

Escreva um programa C que implemente esta brincadeira conforme descrito a seguir.

Entrada

A primeira linha da entrada contém um inteiro N ($N \geq 1$) representando a quantidade de números inteiros que Alan deve responder *yes* (*capicua*) ou *no*. Cada uma das N linhas seguintes será composta por um inteiro de até quatro algarismos.

Observação: O seu programa não poderá decompor o número na entrada, ou seja, não poderá ler o número N como caracteres individuais que formam o número: você deve fazer o programa lê-lo como número e, em seguida, manipulá-lo da maneira que desejar para que possa gerar a resposta correta.

Saída

A saída consiste de uma única linha: a sequência de palavras *yes/no*, separadas por um único espaço em branco entre cada par consecutivo delas, que corresponde à resposta.

Exemplos

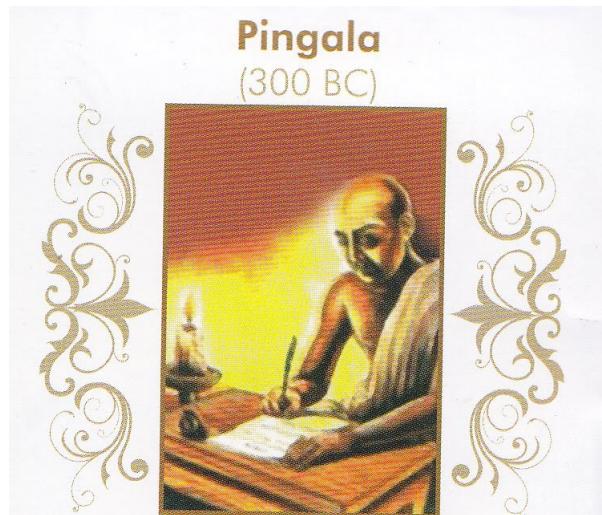
Entrada	Saída
2 4569 5775	no yes

Entrada	Saída
3 1458 1228 9779	no no yes

Entrada	Saída
4 1221 2222 3113 7887	yes yes yes yes

Entrada	Saída
5 1234 1243 1324 2134 4321	no no no no no

Entrada	Saída
10 1234 5897 1881 2662 5588 9229 3653 5555 3773 4587	no no yes yes no yes no yes yes no



5 Computação (++)



(++)

A capacidade natural do ser humano para calcular quantidades, nos mais variados modos, foi um dos fatores que possibilitaram o desenvolvimento da Matemática, da Lógica e, por conseguinte, da Computação. Nos primórdios da Matemática e da Álgebra, utilizavam-se os dedos das mãos para efetuar cálculos, daí a origem da palavra *dígito*.

Por volta do século III a.C., o matemático indiano Pingala inventou o sistema de numeração binário, que ainda hoje é utilizado no processamento de todos computadores digitais: o sistema estabelece que sequências específicas de 1's (uns) e 0's (zeros) pode representar qualquer número, letra, imagem, etc.

Entretanto, a Computação está evoluindo rapidamente e recentemente a SBC (Sociedade Brasileira de Computação) inventou um computador com a base 4 (tetrade), inspirado na Biologia (lembra-se? Adenina, Citosina, Guanina e Timina, as quatro bases nitrogenadas!).

A SBC contratou você para fazer um programa C que receba um número inteiro positivo, na base decimal, e converta-o para a base 4 utilizando divisões sucessivas. Você deve escrever um programa que, a partir de uma lista de números, calcule o valor correspondente de cada desses números na base 4.

Observação: Considere que os símbolos utilizados para representar as quantidades ZERO, UM, DOIS e TRÊS, na base 4 são, respectivamente, A, C, G e T.

Entrada

A entrada contém um único conjunto de testes, que deve ser lido do dispositivo de entrada padrão (o teclado).

A primeira linha contém o número de inteiros N ($1 \leq N \leq 100$) que será digitada.

A segunda linha contém N números inteiros n_i , cada um representando um número decimal.

Saída

Seu programa deve imprimir, na saída padrão, os valores correspondentes na base 4 – um por linha – para cada número decimal digitado.

Exemplos

Observação: Note que as letras (A, C, G, T) são sempre grafadas em maiúsculas.

Entrada	Saída
5 1 2 3 4 10	C G T CA GG

Entrada	Saída
2 16 8	CAA GA

Entrada	Saída
9 1 2 3 4 10 16 8 5 11	C G T CA GG CAA GA CC GT

Entrada	Saída
5 10 20 30 40 50	GG CCA CTG GGA TAG

Entrada	Saída
10 1 2 3 4 1 2 3 4 1 2	C G T CA C G T CA C G



6 Envelopes (++)



(++)

Kurt Gödel é um garoto muito esperto e que adora promoções e sorteios. Como já participou de muitas promoções da forma “para participar, envie n rótulos de produtos ...”, Kurt tem o hábito de guardar o rótulo de todos os produtos que compra prevendo a possibilidade de ocorrência de promoções futuras. Dessa forma, sempre que uma empresa faz uma promoção ele já tem “*um monte*” de rótulos para mandar. A SBC (Super Balas e Caramelos Ltda) está fazendo uma nova SUPER promoção, e, como era de se esperar, Kurt quer participar dela: é preciso enviar um envelope contendo um rótulo de cada tipo de bala que a SBC produz.

Por exemplo, se a SBC disser que produz três tipos de balas (A, B, C) e uma pessoa tem três rótulos de A, três rótulos de B e dois rótulos de C, ela pode enviar no máximo dois envelopes, já que falta um rótulo de C para compor o terceiro envelope.

Sabe-se que não há limite para o número de envelopes que uma pessoa pode enviar para a promoção da SBC. Balas são a segunda coisa de que Kurt mais gosta (a primeira, como você, sabe são as *promoções*) e, por causa disso, a quantidade de rótulos que ele tem é muito grande: ele não está conseguindo determinar a quantidade máxima de envelopes que pode enviar.

Como você é o(a) melhor amigo(a) de Kurt, ele pediu sua ajuda para fazer o cálculo, de modo que ele compre o número exato de envelopes necessários para enviar para esta promoção.

Você deve escrever um programa C que, a partir da lista de rótulos de Kurt, calcule o número máximo de envelopes válidos que ele pode enviar para a promoção da SBC.

Entrada

A entrada contém um único conjunto de testes, que deve ser lido do dispositivo de entrada padrão (o teclado).

A primeira linha contém dois números inteiros N e K representando, respectivamente, a quantidade de rótulos de balas que Kurt possui e o número de tipos diferentes de bala que a SBC produz. Os tipos de balas são identificados por inteiros de 1 a K , com $1 \leq N \leq 1000$ e $1 \leq K \leq 20$.

A segunda linha contém N números inteiros, digamos $n_i \in \{1, 2, 3, \dots, K\}$, cada um representando um rótulo de bala que Kurt possui.

Saída

Seu programa deve imprimir, na saída padrão, o número máximo de envelopes válidos que Kurt pode enviar.

Exemplos

Entrada	Saída
10 2 1 1 1 1 1 2 2 2 2 2	5

Entrada	Saída
20 5 1 2 3 4 1 2 3 4 1 2 3 4 5 1 2 3 4 5 4 4	2

Entrada	Saída
10 3 1 2 3 1 2 3 1 2 3 1	3

Entrada	Saída
20 1	20



7 sapos (+++)



(+++)

Sebastião Bueno Coelho, apelidado de SBC pelos familiares e amigos, passou as férias de janeiro de 2019 no sítio de seus avós. Durante sua estadia, uma das atividades prediletas do SBC era nadar no rio que havia no “fundo” da casa dos avós. Uma das características do rio que mais impressionava SBC era um belo caminho, feito inteiramente com pedras brancas.

Há muito tempo, o avô de SBC notara que os habitantes do sítio atravessavam o rio com grande frequência e, por isso, construiu um caminho nele feito com pedras posicionadas em linha reta; ao fazê-lo, tomou muito cuidado para que o espaçamento entre as pedras fosse exatamente de um metro. Hoje em dia, a única utilidade do caminho é servir de diversão para os sapos que vivem no rio, que pulam de uma pedra a outra agitadamente.

Um certo dia, enquanto descansava e nadava nas águas, SBC assistiu atentamente às acrobacias dos anfíbios Anura e notou que cada sapo sempre pulava uma quantidade fixa de metros.

SBC sabe que você participa, todos os anos, da *Maratona de Programação* do INF/UFG capitaneada pelo Prof. Humberto Longo, do INF/UFG, e chegando na escola, resolveu desafiar-lhe com o seguinte problema:

“Dado o número de pedras no rio – P –, o número de sapos – S , a pedra inicial sobre a qual cada sapo está, sabendo-se que cada pedra é identificada por sua posição na sequência de pedras a partir da margem do rio que está no “fundo” da casa dos avós de SBC – $1, 2, 3, \dots$ – e, por fim, a distância que cada sapo pula – $d_1, d_2, d_3, \dots, d_p$ –, determinar as posições onde pode existir pelo menos um sapo depois que SBC chega no rio após assistir ao balé dos pulos dos sapos.”.

Entrada

A primeira linha da entrada contém dois inteiros P ($1 \leq P \leq 50$) e S ($1 \leq S \leq 100$) representando, respectivamente, o número de pedras no rio e o número de sapos.

Cada uma das S linhas seguintes possui dois inteiros p_i e d_i representando, respectivamente, a posição inicial de um sapo i e a distância fixa de pulo dele.

Saída

A saída contém P linhas. A j -ésima linha indica a possibilidade, ou não, de ter um sapo na j -ésima pedra.

Para as pedras que podem ter um sapo você deve imprimir 1, e para as pedras que, com certeza, não podem ter nenhum sapo você deve imprimir 0.

Exemplos

Entrada	Saída
5 2	1
3 2	0
4 4	1
	1
	1

Entrada	Saída
8 3	0
3 3	1
2 2	1
6 2	1
	0
	1
	0
	1

Entrada	Saída
10 8	1
1 7	1
2 5	1
3 4	1
4 7	1
5 2	1
6 9	1
7 2	1
8 3	1
	0

Entrada	Saída
16 7	1
1 8	1
2 7	1
3 6	1
4 5	1
5 4	1
6 3	1
7 2	0
	1
	0
	1
	1
	1
	1
	1
	1
	1

Entrada	Saída
10 10	1
1 1	1
2 1	1
3 1	1
4 1	1
5 1	1
6 1	1
7 1	1
8 1	1
9 1	1
10 1	1



8 primos (++)



(++)

No livro *A música dos números primos*, de Marcus du Saboy (2007, Editora Zahar, 471 páginas), o autor mostra que o mistério dos *números primos* passou a ser considerado o maior problema matemático de todos os tempos. Em meados do século XIX, o alemão Georg Friedrich Bernhard Riemann (1826 – 1866) formulou uma hipótese:

“É possível estabelecer uma harmonia entre esses números primos, à semelhança da harmonia musical.” A partir de então, as mentes mais ambiciosas da Matemática embarcaram nessa procura que parece não ter fim. Atualmente, estipulou-se o prêmio de um milhão de dólares para quem provar a hipótese. O livro relata esse verdadeiro *Santo Graal* da Matemática, com casos interessantes e retratos pitorescos dos personagens que, desde Euclides, se envolveram nesse estranho mistério.

Você deverá, assim, pesquisar e implementar, em C, um algoritmo que seja capaz de identificar se um dado número inteiro positivo é, ou não, um *número primo*. Número que não é primo é denominado de *composto*.

Entrada

A primeira linha da entrada contém um inteiro N ($1 \leq N \leq 100$) representando a quantidade de números inteiros positivos para os quais seu programa deve responder *primo* ou *composto*.

Cada uma das N linhas seguintes será composta por um inteiro positivo.

Observação: O seu programa deve estar preparado para receber números no intervalo de 2 a $2^{64} - 1$.

Saída

A saída consiste de N linhas, cada uma com a palavra *primo*, se o número for primo, ou a palavra *composto* caso o número não seja primo (número composto). Note que as palavras devem ser grafadas, necessariamente, com letras minúsculas.

Exemplos

Entrada	Saída
5	primo
2	primo
3	primo
11	composto
16	composto
60	

Entrada	Saída
9	composto
1200	primo
1697	composto
2712	primo
2549	primo
4723	primo
7853	primo
23557	composto
23558	primo
15485863	

Entrada	Saída
10	primo
23	primo
29	primo
31	primo
37	primo
73	primo
79	primo
83	primo
101	primo
103	primo
107	primo

Entrada	Saída
10	composto
24	composto
30	composto
32	composto
38	composto
74	composto
80	composto
84	composto
102	composto
104	composto
108	composto



9 vetores (+++)



Uma operação comum em diversas áreas da computação científica é a multiplicação de números inteiros positivos com grande número de dígitos. Por exemplo, multiplicar um número de 24 dígitos por outro de 16 dígitos, o que pode gerar um número de até 40 dígitos.

Você está participando de uma equipe de desenvolvimento de uma aplicação científica que deve implementar, utilizando o conceito de *vetor* para representar cada um dos números envolvidos, a operação de multiplicação mencionada.

A aplicação deve ser desenvolvida utilizando a linguagem C, conforme a seguir especificado.

Entrada

A primeira linha da entrada conterá o número de casos de teste, t , a serem aplicados. Sabe-se que $1 \leq t \leq 50$.

A seguir são apresentadas t linhas, cada uma contendo os dois números inteiros a serem multiplicados, digamos m e n , sabendo-se que eles terão no máximo 40 dígitos cada, mas que também poderão ser iguais a 0 (zero). A dupla de números está separada por um único espaço em branco.

Saída

A saída consiste de t linhas, cada uma com o resultado da operação de multiplicação dos pares de números correspondentes, na ordem em que foram fornecidos.

Exemplos

Entrada	Saída
1 9423891297239 123857601272	1167220570724098896488008

Observação: Devido ao comprimento dos números envolvidos, os exemplos podem ter mais linhas impressas que aquelas registradas nos dados de entrada fornecidos (veja o 2º exemplo). Sempre considere que cada *caso* é fornecido numa única linha, com os números m e n sendo fornecidos numa única linha e separados por um único espaço em branco entre eles.

Entrada	Saída
2 9423891297239 123857601272 737238112845712940348123 1934720871365475	1167220570724098896488008 1426349964088696127858578510648863253425

Entrada	Saída
6 0 104759 0 104801 0 105331 104743 0 104789 0 104987 0	0 0 0 0 0 0

Entrada	Saída
2 17546 92130935 737238112845712940348123 1934720871365475	1616529385510 1426349964088696127858578510648863253425

Entrada	Saída
2 0 92130935 737238112845712940348123 0	0 0



10 cálculo de áreas (+++)



(++)

Um grande amigo(a) seu(sua), dos tempos de colégio, está cursando Arquitetura na UFG e pediu auxílio para você para resolver o seguinte problema:

Ele precisa calcular a área, em metros quadrados, de diversas figuras planas:

C círculo – cujo raio é dado por R ;

E elipse – cujos raios maior e menor são, respectivamente, R e r ;

T triângulo – cujos lados são a , b e c (nesta ordem);

Z trapézio – cujas bases maior e menor são, respectivamente, B e b , e a altura é H (nesta ordem).

Considera-se que vocês conhecem as “fórmulas matemáticas” de cálculo para as áreas destas figuras, mas você pensou numa solução mais sofisticada: elaborar um programa de computador C que seja capaz de receber as informações necessárias e retornar a área da figura.

Observação: Utilize $\pi = 3,14159265$.

Entrada

A primeira linha da entrada contém um inteiro N ($N \geq 1$) representando a quantidade de figuras planas para os quais seu programa deve calcular as áreas.

Cada uma das N linhas seguintes será composta por, primeiramente, um caractere que identifica qual é a figura e, em seguida, os parâmetros necessários para calcular sua área, na ordem anteriormente especificada e sempre separados por um único espaço em branco entre eles. Os parâmetros serão sempre números inteiros exritante positivos.

Saída

A saída consiste de N linhas, cada uma contendo a área da respectiva figura plana, com quatro casas decimais de precisão.

Observação: Considere que a resposta (área de cada figura plana), de acordo com seu(sua) amigo(a), precisa ter somente a parte inteira podendo, portanto, ser desprezada da parte decimal, desde que devidamente arredondada.

Exemplos

Entrada	Saída
4 C 2 E 2 4 T 8 8 8 Z 7 3 4	13 25 28 20

Entrada	Saída
4 C 5 E 3 7 T 3 4 5 Z 7 10 4	79 66 6 34

Entrada	Saída
3 T 3 4 5 T 5 5 5 T 6 8 10	6 11 24

Entrada	Saída
3 E 5 5 E 4 8 E 1 2	79 101 6

Entrada	Saída
3 T 2 2 2 T 4 4 4 T 9 9 9	2 7 35



11 manipulação de matrizes 1 (++)



(++)

Um fundamental conceito abstrato da Matemática, extremamente utilizado em Computação, é o de *matriz*.

Uma matriz pode ser unidimensional (um vetor), bidimensional, tridimensional, etc.

Considerando apenas as matrizes bidimensionais A , de ordem m por n , que armazenam em cada uma de suas posições um número inteiro $a_{i,j}$, onde o índice i indica a linha e o índice j indica a coluna, com $1 \leq i \leq m$ e $1 \leq j \leq n$ e $m, n \in \mathbb{N}^*$, escreva um programa C que atenda às especificações indicadas a seguir.

Entrada

A primeira linha da entrada contém os números naturais m e n , nesta ordem, separados por um único espaço em branco entre eles.

Cada uma das m linhas seguintes conterão os elementos localizados em cada uma das linhas da matriz A , separados entre si por um único espaço em branco. Sabemos, portanto, que cada uma destas linhas conterá n números inteiros.

A linha seguinte conterá um único caractere que indicará uma operação matricial: poderá ser o '+' (mais) para indicar a *adição* ou o 'x' (xis) para indicar a *multiplicação*.

Por fim, as últimas m linhas da entrada conterão os elementos localizados em cada uma das linhas da matriz B , separados entre si por um único espaço em branco. Sabemos, portanto, que cada uma destas linhas conterá n números inteiros.

Observação: Considere que $1 \leq m, n \leq 10$ e que $-50 \leq a_{i,j}, b_{i,j} \leq 50$.

Saída

A saída consistirá das linhas da matriz que corresponda à realização da operação $A + B$ ou $A \times B$.

Lembre-se que há regras específicas para que a operação de multiplicação matricial possa ser realizada, bem como a maneira como esta se processa. Se não for possível realizá-la, o programa deverá emitir na saída uma linha com a mensagem: ERROR (grafada em letras maiúsculas).

Exemplos

Entrada	Saída
2 3 1 2 3 4 5 6 + 6 5 4 3 2 1	7 7 7 7 7 7

Entrada	Saída
2 3 1 2 3 4 5 6 x 6 5 4 3 2 1	ERROR

Entrada	Saída
2 2 1 2 4 5 x 6 5 3 2	12 9 39 30

Entrada	Saída
<pre> 4 4 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 + 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 0 </pre>	<pre> 1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1 </pre>

Entrada	Saída
<pre> 4 4 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 x 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 0 </pre>	<pre> 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 0 </pre>

$$A = \begin{bmatrix} 3 & 5 & 7 & 2 \\ 1 & 4 & 7 & 2 \\ 6 & 3 & 9 & 17 \\ 13 & 5 & 4 & 16 \end{bmatrix} \quad A^{-1} = \begin{bmatrix} 15 & 21 & 0 & 15 \\ 23 & 9 & 0 & 22 \\ 15 & 16 & 18 & 3 \\ 24 & 7 & 15 & 3 \end{bmatrix}$$

$$\det(A) = 21 \quad \det(A^{-1}) = 5$$

12 manipulação de matrizes 2 (++++)



(++++)

A partir do exercício anterior e, portanto, continuando com a proposta de considerar apenas as matrizes bidimensionais A , de ordem m por n , que armazenam em cada uma de suas posições um número inteiro $a_{i,j}$, onde o índice i indica a linha e o índice j indica a coluna, com $1 \leq i \leq m$ e $1 \leq j \leq n$ e $m, n \in \mathbb{N}^*$, escreva um programa \mathbb{C} que atenda às especificações indicadas a seguir.

Entrada

A primeira linha da entrada contém os números naturais m e n , nesta ordem, separados por um único espaço em branco entre eles.

Cada uma das m linhas seguintes conterão os elementos localizados em cada uma das linhas da matriz A , separados entre si por um único espaço em branco. Sabemos, portanto, que cada uma destas linhas conterá n números inteiros.

A linha seguinte conterá um único caractere que indicará uma operação matricial: poderá ser o 'I' (inversa) para indicar a operação de *inversão de matriz*, 'T' (tê) para indicar a *transposição de matriz* e, por fim, 'D' (dê) para indicar o *determinante* da matriz.

Observação: Considere que $1 \leq m, n \leq 10$ e que $-50 \leq a_{i,j}, b_{i,j} \leq 50$.

Saída

A saída consistirá das linhas da matriz que corresponda à realização da operação A^{-1} (matriz inversa de A), A^t (matriz transposta de A) ou $\det(A)$ (determinante de A).

Lembre-se que há regras específicas que regem a realização das operações de inversão de matriz e do cálculo de seu determinante. Se não for possível realizar a operação solicitada pelo usuário, o programa deverá emitir na saída uma linha com a mensagem: ERROR (grafada em letras maiúsculas).

Exemplos

Entrada	Saída
2 3 1 2 3 4 5 6 T	1 4 2 5 3 6

Entrada	Saída
2 3 1 2 3 4 5 6 D	ERROR

Entrada	Saída
2 2 1 2 4 5 D	-3

Entrada	Saída
4 4 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 I	1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1

Entrada	Saída
4 4 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 0 D	1



13 Números de Fibonacci (++)



(++)

O matemático italiano Leonardo Fibonacci (1170-1250) foi de grande influência na Idade Média, sendo por muitos considerado como o maior deste período. Foi ele quem introduziu na Europa os *números arábicos* e descobriu uma curiosa sequência numérica que, por isso, foi posteriormente batizada de *Sequência de Fibonacci* e os números que a formam de *Números de Fibonacci*.

Aos 32 anos, Fibonacci publicou o livro *Liber Abaci* (ou seja, o *Livro do Ábaco* ou *Livro de Cálculo*), responsável pela disseminação dos números hindu-arábicos na Europa.

Como ele prestou grandes serviços à cidade de Pisa há nela uma estátua em sua homenagem, localizada na galeria ocidental do Camposanto (mostrada no cabeçalho desta questão).

Os *Números de Fibonacci* são definidos da seguinte maneira:

$$f_0 = 0$$

$$f_1 = 1$$

$$f_2 = 1$$

$$f_n = f_{n-1} + f_{n-2}, \text{ com } n \in \mathbb{N} \text{ e } n \geq 3$$

Escreva, em C, um programa que receba o valor de n conforme anteriormente definido, $3 \leq n \leq 100$, e escreva na saída o valor de f_n correspondente.

Entrada

A primeira linha da entrada contém um número inteiro k , $1 \leq k \leq 10$, que corresponde ao número de casos de teste que serão fornecidos nas linhas seguintes. Cada uma destas linhas conterá um valor específico

para n .

Saída

Seu programa deve imprimir k linhas, cada uma contendo o valor calculado para o n correspondente na entrada.

Exemplos

Entrada	Saída
4	2
3	3
4	5
5	8
6	

Entrada	Saída
7	13
7	21
8	34
9	55
10	89
11	144
12	233
13	

Observação: Lembre-se que um *Número de Fibonacci* pode ser extremamente grande. Por exemplo, $f_{100} = 354224848179261915075$. Portanto isto deve ser previsto no seu programa.

1296	2
648	2
324	2
162	2
81	3
27	3
9	3
3	3
1	

14 Fatoração de Números de Fibonacci (++++)



(++++)

Dando sequência ao estudo acerca dos *Números de Fibonacci*, o que se deseja agora é apresentar a *fatoração* de um certo número destes.

Você deverá escrever, novamente em C, um programa que receba o valor de k e, em sequência, um conjunto de valores n e imprima a fatoração de cada um dos f_n solicitados.

Entrada

A primeira linha da entrada contém o número inteiro k , $1 \leq k \leq 10$, que corresponde ao número de casos de teste que serão fornecidos nas linhas seguintes. Cada uma destas linhas conterá um valor para n , $1 \leq k \leq 100$.

Saída

Seu programa deve imprimir k linhas, cada uma contendo os fatores de f_n , apresentados em ordem estritamente crescente e separados por um espaço em branco.

Exemplos

Observação: No último exemplo tem-se que $f_{100} = 354224848179261915075$. Sua fatoração é:
 $3 \times 5 \times 5 \times 11 \times 41 \times 101 \times 151 \times 401 \times 3001 \times 570601$

Entrada	Saída
4 3 4 5 6	2 3 5 2 2 2

Entrada	Saída
7 7 8 9 10 11 12 13	13 3 7 2 17 5 11 89 2 2 2 2 3 3 233

Entrada	Saída
1 100	3 5 5 11 41 101 151 401 3001 570601

Universidade Federal de Goiás – UFG
Instituto de Informática – INF
Bacharelados (Núcleo Básico Comum)

Algoritmos e Estruturas de Dados 1 – 2020/2

Lista de Exercícios nº 02 – Recursividade
(Turmas: INF0061/INF0286 – Prof. Wanderley de Souza Alencar)

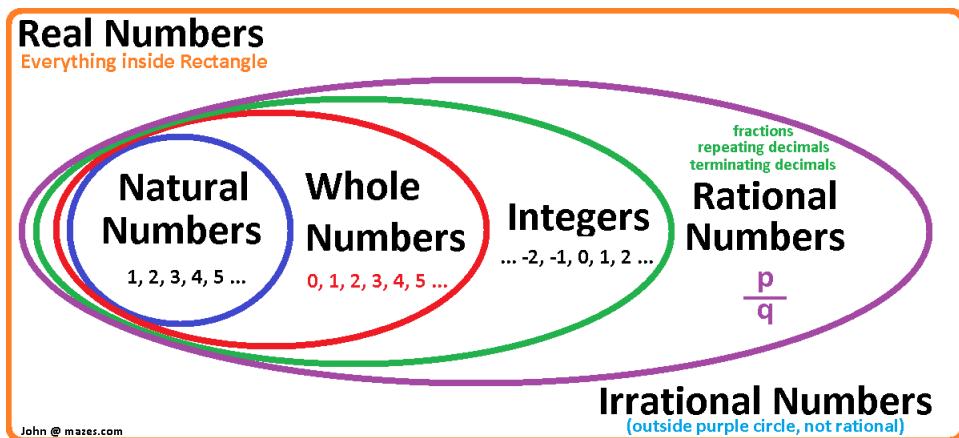
Sumário

1 Imprimindo números naturais recursivamente	2
2 Sequência de Fibonacci recursiva	4
3 Função de Ackermann	8
4 Reverso de um número natural	10
5 Conversão de decimal para binário	12
6 Fatorial duplo	14
7 O banco inteligente	16
8 Famílias de Tróia	19
9 Torre de Hanoi	21
10 Setas	23
11 Batalha naval	26
12 Labirinto – 1	29
13 Labirinto – 2	31
14 Pegar e escapar	33
15 Altas aventuras	35

1 Imprimindo números naturais recursivamente



(+)



Os *números naturais* são os números utilizados ordinariamente para contagem:

$$\mathbb{N}^* = \{1, 2, 3, \dots\}$$

e, por isso, às vezes são chamados de *números de contagem*. Eles são ditos *naturais* devido à nossa experiência natural, geralmente na infância, em que apenas manipulamos quantidades discretas de objetos: uma balinha, dois chiquetes, um pedaço de bolo, e certa quantidade de outras guloseimas. Ou, ainda, quando reclamávamos de ter “*muitas tarefas*” que a professora havia “*passado*” para casa – em verdade eram apenas três pequenos exercícios!

Ao matemático alemão Leopold Kronecker (1823 – 1891) está associada a seguinte frase:
“Deus criou os números naturais; o resto é obra do homem.”.

Por algum tempo houve polêmica quanto ao numeral 0 (zero) pertencer, ou não, aos números naturais, já que, habitualmente, não se inicia uma contagem pelo valor “zero”. Entretanto ele representa um conceito importante: a ausência de elementos num conjunto, seja ele abstrato ou concreto.

A Matemática contemporânea representa o conjunto destes números por meio do símbolo \mathbb{N} , incluindo o 0 (zero). Para excluí-lo utiliza-se o asterisco como expoente: \mathbb{N}^* , como feito no exemplo inicial desta questão. A partir deste conceito inicial a respeito dos números naturais, deseja-se que você escreva um programa, em \mathbb{C} , para imprimir os n primeiros números naturais usando o conceito de recursividade, que os define da seguinte maneira:

$$n_0 = 0$$
$$n_{i+1} = n_i + 1, i \in \{0, 1, 2, \dots\}$$

Entrada

A única linha da entrada contém um único natural n , indicando que se deseja imprimir os n primeiros números naturais, sendo que $n \in \mathbb{N}^*$ e $n \leq 5000$.

Saída

Seu programa deve imprimir uma única linha, contendo os n primeiros números naturais separados por um único espaço em branco entre eles.

Exemplos

Entrada	Saída
37	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37

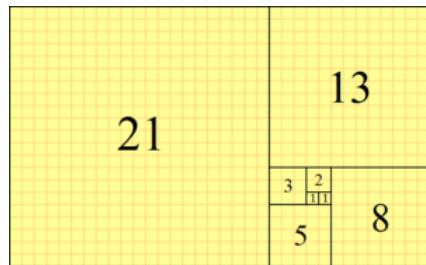
Entrada	Saída
50	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50

Observação: Nos exemplos anteriores a saída *parece* ocupar mais de uma linha devido à restrição de largura da página impressa. Apesar disso, considere que todos os números apresentados na saída estão *numa única linha* da saída.

2 Sequência de Fibonacci recursiva



(+)



Sem dúvida a chamada “Sequência de Fibonacci” (ou “Sucessão de Fibonacci”) é uma das mais famosas sequências numéricas da Matemática. Os dois primeiros termos desta sequência são:

$$f_0 = 0 \quad f_1 = 1.$$

A partir do terceiro termo, cada termo é obtido somando-se os dois termos imediatamente anteriores a ele, ou seja:

$$f_n = f_{n-1} + f_{n-2}, \text{ com } n \in \mathbb{N} \text{ e } n \geq 2$$

Portanto, os seus dez primeiros termos são 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

Muitas fontes históricas registram que ela foi “descoberta” (ou “inventada”) por Leonardo Fibonacci (1170 – 1250), um matemático italiano que também ficou conhecido como Leonardo de Pisa, sua cidade de nascimento, pois apenas no século XIX ficou associado ao nome *Fibonacci* que, de maneira aproximativa, significa “o filho de Bonacci”, em referência ao seu pai Guglielmo dei Bonacci, um próspero mercador.

Aos 32 anos, em 1202, Fibonacci publicou o livro *Liber Abaci* (Livro do Ábaco, ou Livro de Cálculo), um livro de receitas a respeito de como realizar cálculos e que foi o responsável pela disseminação dos números hindu-arábicos na Europa. Num trecho desta obra, Leonardo introduz a sequência por meio de um problema envolvendo coelhos. O problema dizia que:

“Iniciando com um par de coelhos – um macho e uma fêmea – depois de um mês eles se tornam sexualmente adultos e produzem um par de filhotes, também um macho e uma fêmea. Novamente, um mês depois, estes coelhos reproduzem e geram outro par macho-fêmea, os quais, por sua vez, também gerarão outro par macho-fêmea depois de um mês (Claro: ignore aqui a pequeníssima probabilidade de que isto efetivamente ocorra no mundo natural dos coelhos.)

A questão é: depois de um ano, quantos coelhos haverá?”

A resposta ao problema é obtida por meio do uso da Sequência de Fibonacci – veja “*The 11 most beautiful Mathematical equations*” em:

<https://www.livescience.com/57849-greatest-mathematical-equations.html>.

Uma curiosidade é que, depois disso, Leonardo nunca mais citou a sequência, que ficou esquecida até o século XIX quando matemáticos que trabalhavam em propriedades de sequências numéricas a recuperaram, cabendo ao matemático francês Édouard Lucas (1842 – 1891) ter nomeado, oficialmente, o problema dos coelhos com o nome de “Sequência de Fibonacci”.

Entretanto, sabe-se hoje, que Leonardo não descobriu ou inventou a famosa sequência (veja a obra de Keith Devlin intitulada *Finding Fibonacci: The Quest to Rediscover the Forgotten Mathematical Genius*

Who Changed the World, Princeton University Press, 2017), pois citações a ela aparecem em textos antigos, em Sânsrito, muito antes da sua menção por Leonardo.

Vamos a um problema envolvendo-a:

Considere que seja dado um número n , $n \in \mathbb{N}^*$. Usando o conceito de recursividade, elabore um programa em C para imprimir até o n -ésimo termo da “Série de Fibonacci”.

Observação: Note que a contagem dos termos foi iniciada com o termo 0 (zero): $f_0 = 0$.

Entrada

A única linha da entrada contém um número natural n , indicando a ordem máxima dos termos desejados da “Série de Fibonacci”. Sabe-se que $1 \leq n \leq 1000$.

Saída

Seu programa deve imprimir uma única linha contendo até o n -ésimo termo da série, sempre separados por um único espaço em branco.

Exemplos

Entrada	Saída
0	0

Entrada	Saída
1	0 1

Entrada	Saída
8	0 1 1 2 3 5 8 13 21

Entrada	Saída
11	0 1 1 2 3 5 8 13 21 34 55 89

Entrada	Saída
15	0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610

Observação: Uma questão interessante é:

Será possível encontrar uma *fórmula* explícita que seja capaz de fornecer um determinado termo da “Sequência de Fibonacci” sem a necessidade de realizar a geração de todos os termos anteriores?

Se isto for possível, gerar utilizar a *fórmula* será mais *eficiente* que utilizar uma função geradora, seja ela recursiva ou iterativa?

Se não for possível, qual o motivo da *impossibilidade*?



3 Função de Ackermann



(+)

Na teoria da computabilidade, a *Função de Ackermann* (f_{ack}), nomeada por Wilhelm Friedrich Ackermann (1896 – 1962), é um dos mais simples exemplos de uma função computável que não é função recursiva primitiva. Todas as funções recursivas primitivas são totais e computáveis, mas a *Função de Ackermann* mostra que nem toda função total-computável é recursiva primitiva.

Depois que Ackermann publicou sua função (que continha três números naturais como argumentos), vários autores a modificaram para atender a diversas finalidades. Então, a f_{ack} pode ser referenciada a uma de suas várias formas da função original.

Uma das versões mais comuns, a *Função de Ackermann-Péter*, que possui apenas dois argumentos, é definida a seguir para números naturais m e n :

$$f_{ack}(m, n) = \begin{cases} (n + 1), & \text{se } m = 0 \\ f_{ack}(m - 1, 1), & \text{se } n = 0, m > 0 \\ f_{ack}(m - 1, f_{ack}(m, n - 1)), & \text{se } n > 0, m > 0 \end{cases}$$

Entrada

A única linha da entrada contém dois números naturais m e n separados por um único espaço em branco, nesta ordem, representando os parâmetros para a *Função de Ackermann*.

Saída

Seu programa deve imprimir uma única linha com o valor da f_{ack} para os dois parâmetros recebidos.

Exemplos

Entrada	Saída
0 7	8

Entrada	Saída
3 0	5

Entrada	Saída
3 2	29

Entrada	Saída
2 4	11

4 Reverso de um número natural



(+)



Todo número natural estritamente positivo $n \in \mathbb{N}^*$ possui um *número reverso* correspondente. Por exemplo, considere que n seja escrito da seguinte maneira:

$$n = d_k d_{k-1} d_{k-2} \cdots d_2 d_1 d_0$$

onde $k \in \mathbb{N}^*$ corresponde ao número de dígitos significativos que formam n , ou seja, $d_k \in \{1, 2, 3, \dots, 9\}$ e $d_i \in \{0, 1, 2, \dots, 9\}$, com $0 \leq i < k$.

O *número reverso* de n é $n^r = d_\ell d_{\ell-1} d_{\ell-2} \cdots d_{k-2} d_{k-1} d_k$, sendo d_ℓ o primeiro dígito não nulo, tomados nesta ordem, dentre $d_k d_{k-1} d_{k-2} \dots d_2 d_1 d_0$ do número original n .

Escreva uma função recursiva, em \mathbb{C} , que seja capaz de determinar o *número reverso* de um certo número natural estritamente positivo n fornecido como entrada.

Entrada

A única linha da entrada contém um único número natural estritamente positivo, n , $1 \leq n \leq 10^6$.

Saída

Seu programa deve imprimir uma única linha com o valor de n^r , o *número reverso* de n .

Exemplos

Entrada	Saída
411	114

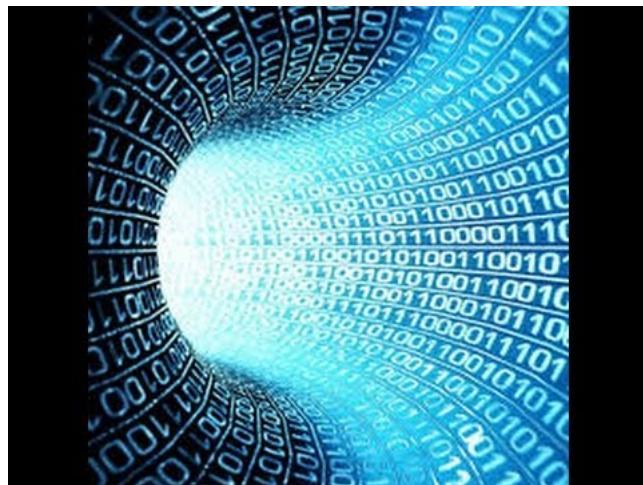
Entrada	Saída
1230	321

Entrada	Saída
138000	831

5 Conversão de decimal para binário



(++)



Escreva um programa, em C, que receba um número natural $n \in \mathbb{N}$, representado utilizando a notação decimal, e o converta para sua notação binária. O programa deve utilizar uma “*função recursiva*” para realizar a conversão.

Entrada

A primeira linha conterá um número natural estritamente positivo k , $1 \leq k \leq 1000$, que representa o número de casos de teste que virão em seguida.

Cada uma das k linhas seguintes possuem, cada uma, um único número natural, $0 \leq n_i < 10^6$, com $1 \leq i \leq k$, representado utilizando a notação decimal, a ser convertido para sua correspondente representação binária.

Saída

Seu programa deve imprimir k linhas, cada uma com a correspondente representação binária de um número da entrada.

Exemplos

Entrada	Saída
5	1
1	10
2	11
3	100
4	101
5	

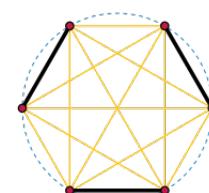
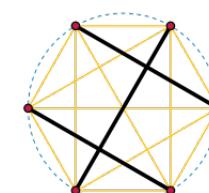
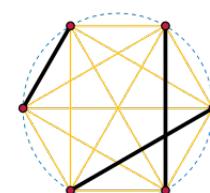
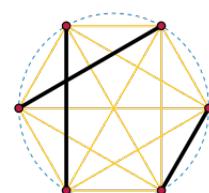
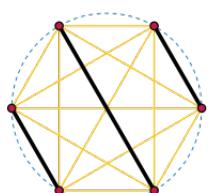
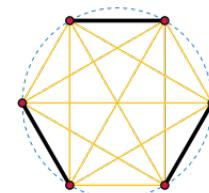
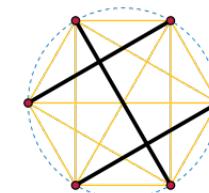
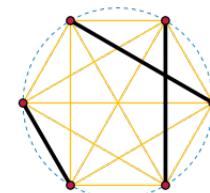
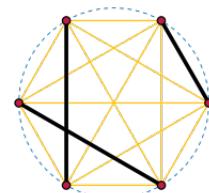
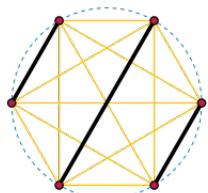
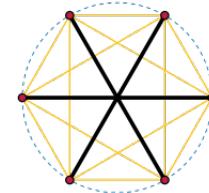
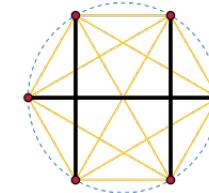
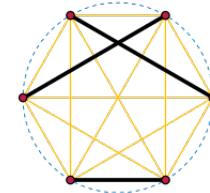
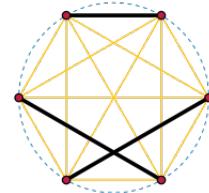
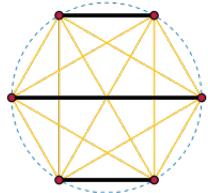
Entrada	Saída
3 321 753 255	101000001 1011110001 11111111

Entrada	Saída
1 373728	1011011001111100000

6 Fatorial duplo



(++)



Pode-se definir uma função $\ddot{f}(n)$, *fatorial duplo* de n , com $n \in \mathbb{N}$, como sendo o produto de todos os números naturais ímpares de 1 até n , inclusive este, quando ele é ímpar. Assim, por exemplo, tem-se que:

$$\ddot{f}(1) = 1$$

$$\ddot{f}(2) = 1$$

$$\ddot{f}(3) = 3$$

$$\ddot{f}(5) = 15$$

Você deve escrever uma função recursiva, em \mathbb{C} , que seja capaz de, recebendo n , imprimir o valor de $\ddot{f}(n)$.

Entrada

A única linha de entrada contém o valor de n , com $1 \leq n \leq 100$.

Saída

Imprima uma única linha de saída, com o valor de $\ddot{f}(n)$.

Exemplo

Entrada	Saída
1	1

Entrada	Saída
7	105

Entrada	Saída
10	945

7 O banco inteligente



(+++)



Os atuais caixas automáticos dos bancos, ou ATMs – *Automated Teller Machines*, são uma ótima invenção mas, às vezes, precisamos de dinheiro *trocado* e a máquina nos entrega notas de R\$100,00. Noutras vezes, desejamos sacar um valor um pouco maior e, por questão de segurança, gostaríamos de receber todo o valor em notas de R\$100,00, mas a máquina nos entrega um *monte* de notas de R\$20,00.

Para conquistar clientes, o Banco Inteligente (BI) está tentando minimizar este problema dando aos clientes a possibilidade de escolher o valor das notas na hora do saque. Para isso, eles precisam da sua ajuda para saber a resposta para a seguinte questão: dado um determinado valor S de saque (em reais) e quantas notas de cada valor a máquina tem, qual é o número de maneiras distintas que há para entregar o valor S ao cliente?

Sabe-se que nas ATMs do BI há escaninhos para notas de 2, 5, 10, 20, 50 e de 100 reais.

Por exemplo, suponha que para certo cliente X tenha-se que $S = 22$ e que o número de notas de cada valor presente na ATM no momento da solicitação deste saque é:

$$N_2 = 5$$

$$N_5 = 4$$

$$N_{10} = 3$$

$$N_{20} = 10$$

$$N_{100} = 10$$

(1)

Assim, há QUATRO maneiras distintas da máquina entregar o valor do saque solicitado:

1^a : uma nota de R\$20,00 e uma nota de R\$2,00 (duas notas);

2^a : duas notas de R\$10,00 e uma nota de R\$2,00 (três notas);

3^a : uma nota de R\$10,00, duas notas de R\$5,00 e uma nota de R\$2,00 (quatro notas);

4^a : quatro notas de R\$5,00 e uma nota de R\$2,00 (cinco notas).

Tarefa

Escrever, em C, um programa que seja capaz de determinar o número de maneiras possíveis de atender à solicitação de saque do cliente.

Entrada

A primeira linha da entrada contém o número natural S expressando, em reais, o valor do saque desejado. A segunda linha contém seis inteiros $N_2, N_5, N_{10}, N_{20}, N_{50}$ e N_{100} , respectivamente, indicando o número de notas de 2, 5, 10, 20, 50 e 100 reais disponíveis na ATM no momento do saque. Os números estão separados por um único espaço em branco entre eles.

Saída

Seu programa deve imprimir um único número natural: a quantidade de maneiras distintas da máquina atender ao saque solicitado.

Restrições

- $0 \leq S \leq 5000$ e $N_i \leq 500, \forall i \in \{2, 5, 10, 20, 50, 100\}$.

Exemplos

Entrada	Saída
22 5 4 3 10 0 10	4

Entrada	Saída
1000 20 20 20 20 20 20	34201

Entrada	Saída
50 1 1 1 1 0 1	0

Observações

Pense como seria alterar este exercício para cada uma das seguintes variações:

1. Considerando o valor S solicitado, a ATM deverá entregar para o usuário o maior número de notas possível para a realização daquele saque. A saída deverá ser o número de notas entregues de cada tipo de cédula, na seguinte ordem: $N_2, N_5, N_{10}, N_{20}, N_{50}$ e N_{100} .

O exemplo nº 01, onde $S = 22$, teria como saída a sequência: 1 4 0 0 0 0. Ou seja, uma nota de R\$2,00 e quatro notas de R\$5,00.

2. Considerando o valor S solicitado, a ATM deverá entregar para o usuário o menor número de notas possível para a realização daquele saque. A saída deverá ser o número de notas entregues de cada tipo de cédula, na seguinte ordem: $N_2, N_5, N_{10}, N_{20}, N_{50}$ e N_{100} .

O exemplo nº 01, onde $S = 22$, teria como saída a sequência: 1 0 0 1 0 0. Ou seja, uma nota de R\$2,00 e uma nota de R\$20,00.

Entrada	Saída
50 2 2 2 2 2	4



8 Famílias de Tróia



(+++)

A *Guerra de Tróia* pode ter sido um grande conflito bélico entre gregos e troianos, possivelmente ocorrido entre os anos de 1.300 a.C. e 1.200 a.C., ou seja, no fim da Idade do Bronze no Mediterrâneo. Recentemente foram encontradas inscrições numa caverna a respeito de sobreviventes deste conflito e, após um trabalho árduo, arqueólogos descobriram que as inscrições descreviam relações de parentesco numa certa população daquela região. Cada item da inscrição indicava duas pessoas que pertenciam a uma mesma família.

O problema dos arqueólogos(as) – que agora é *seu problema* – é determinar quantas famílias distintas existiam naquela população e, obviamente, não eles(as) desejam ter que fazer isto “manualmente”, já que computadores existem para estas atividades de agrupamento, dentre outras.

Entrada

O arquivo de entrada consiste de $(m + 1)$ linhas.

A primeira linha do arquivo de entrada contém dois números naturais n e m . Onde $n, n \in \mathbb{N}$, indica o número de pessoas na população que, por simplicidade, são sempre numeradas de 1 a n e sabe-se que $1 \leq n \leq 5 \times 10^4$. O valor m indica a quantidade de linhas após a primeira linha, sendo que $m \in \mathbb{N}$ e $1 \leq m \leq 10^5$.

As demais m linhas do arquivo de entrada contêm, cada uma, dois números naturais identificando um par de pessoas daquela população por meio de seus números, sempre separados por um único espaço em branco. Cada linha indica que as duas pessoas pertenciam a uma mesma família.

Saída

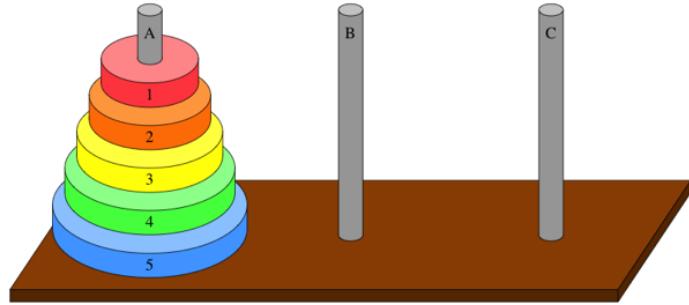
A saída, do programa que será elaborado por você, deve conter apenas uma única linha contendo conte o número de famílias identificadas naquela população.

Exemplos

Entrada	Saída
4 4 1 2 2 3 3 4 4 1	1

Entrada	Saída
8 10 1 2 2 3 3 6 6 5 5 4 4 3 6 7 7 8 8 1 1 5	1

Entrada	Saída
5 4 1 2 2 3 4 5	2



9 Torre de Hanoi



Torre de Hanói é um "quebra-cabeça" que consiste em uma base contendo três pinos (ou hastas), em um dos quais são dispostos alguns discos, uns sobre os outros, em ordem crescente de diâmetro, de cima para baixo. O problema consiste em passar todos os discos de um pino para outro qualquer, usando um dos pinos como *auxiliar*, de maneira que um disco maior nunca fique em cima de outro menor, em nenhuma situação. O número de discos pode variar, sendo que o mais simples contém apenas três discos. (Fonte: Wikipédia)

Suponha que os pinos se chamam “O” (origem), “D” (destino), “A” (auxiliar), faça um programa recursivo que resolva a Torre de Hanói.

Entrada

O arquivo de entrada consiste de uma única linha contendo um número natural n , $n \in \mathbb{N}^* \mid 2 \leq n \leq 1000$, que indica a quantidade de discos contidos no pino de origem – pino “O”. Os discos são, sempre, numerados de 1 a n , indicando o diâmetro do disco (numa determinada unidade de medida qualquer).

Saída

A saída deve conter os movimentos a serem realizados para se resolver a *Torre de Hanói* com os n discos. Cada movimento deve estar em uma linha no formato de par ordenado na forma (*pino de origem*, *pino de destino*), onde *pino de origem* e *pino de destino* $\in \{O, D, A\}$.

Exemplos

Entrada	Saída
2	(O, A) (O, D) (A, D)

Observações

Note que não há “*espaço em branco*” entre as letras indicativas dos pinos nas respostas, como também para os parênteses. Além disso, todas as letras são grafadas em maiúsculas.

Entrada	Saída
3	(O, D) (O, A) (D, A) (O, D) (A, O) (A, D) (O, D)



10 Setas



(++++)

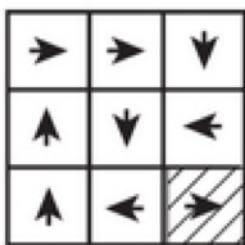
Gabriel é um garoto que gosta muito de um jogo eletrônico onde há várias letras num tabuleiro – que fica sobre o piso – e o jogador precisa, rapidamente, pisar nas letras corretas, de acordo com as instruções que aparecem na *tela de projeção* que está à sua frente, seguindo uma música ao fundo.

Cansado de vencer o “jogo”, Gabriel inventou um novo:

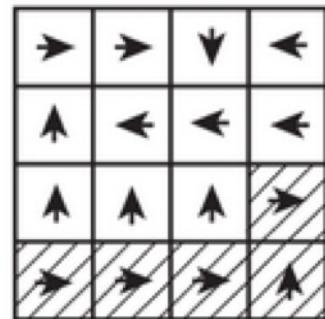
Agora temos um tabuleiro quadrado, com n células de cada lado, em que cada célula possui uma seta que aponta para uma das quatro posições vizinhas ($\blacktriangleright, \blacktriangleleft, \blacktriangleup, \blacktriangledown$). O jogador primeiro escolhe uma célula inicial para se posicionar e, quando a música começa, ele deve caminhar na direção para onde a seta em que ele está naquele momento apontar. Ganhará o jogo quem pisar em mais setas *corretas* durante um determinado período de tempo previamente fixado.

O problema é que Gabriel joga tão rápido que quando a seta atual *manda* ele “sair do tabuleiro”, ele segue a orientação, muitas vezes quebrando alguns objetos próximos ao tabuleiro. Quando isso acontece, dizemos que a célula inicial deste jogo é uma célula *não segura*, pois leva a um caminho que termina fora do tabuleiro.

A figura a seguir mostra dois tabuleiros: um 3×3 e outro 4×4 , respectivamente, com oito e onze células *seguras*:



Tabuleiro 3x3 com oito células seguras



Tabuleiro 4x4 com onze células seguras

As células seguras de cada tabuleiro são as seguintes:

3×3 – todas, exceto a (3, 3);

4×4 – (1,1); (1,2); (1,3); (1,4); (2,1); (2,2); (2,3); (2,4); (3,1); (3,2) e (3,3).

Sua tarefa é ajudar Gabriel: construa um programa C que indique, a partir de uma dada configuração do tabuleiro fornecida, quantas células são *seguras* para ele iniciar o jogo.

Entrada

A primeira linha da entrada contém o número natural n , o tamanho do tabuleiro, com $1 \leq n \leq 500$. Cada uma das n linhas seguintes contém n caracteres, com as direções das setas, sem nenhum espaço entre elas. As direções válidas são:

‘V’ (letra V, maiúscula) aponta para a célula da linha abaixo, na mesma coluna;

‘<’ (sinal menor-que) aponta para a célula à esquerda, na mesma linha;

‘>’ (sinal maior-que) aponta para a célula à direita, na mesma linha;

‘A’ (letra A, maiúscula) aponta para a célula da linha acima, na mesma coluna.

Saída

Seu programa deve produzir um único número natural k : o número de células seguras naquela configuração do tabuleiro.

Exemplos

Entrada	Saída
<pre>3 > > V A V < A < ></pre>	8

Entrada	Saída
4 > > V < A < < < A A A > > > > A	11

Entrada	Saída
4 V > > > V > V < > A > V < < V <	0

Entrada	Saída
5 > > V < < V > V V A V > > > A > > A A < > > A > A	25

11 Batalha naval



(++++)



Pedro e Paulo gostam muito de jogar *Batalha Naval*. Apesar de serem grandes amigos, Pedro desconfia que Paulo não esteja jogando *honestamente* e, para tirar essa dúvida, decidiu usar um programa de computador para verificar o resultado de cada jogo.

Acontece que Pedro não sabe programar e, por isso, pediu a sua ajuda para elaborar este *programa de auditoria naval*, explicando-lhe que cada jogador do jogo *Batalha Naval* possui um tabuleiro retangular com n linhas e m colunas ($n, m \in \mathbb{N}^*$) para representar o “campo de batalha”, onde:

- cada posição é um quadrado que pode conter água ('a') (letra 'a', minúscula) ou uma parte de um navio ('#') (um símbolo *hashtag*);
- dois quadrados são ditos *vizinhos* se possuem um lado comum, ou seja, um lado que pertence a ambos quadrados;
- se duas partes de navio estão em posições vizinhas, então essas duas partes pertencem ao mesmo navio;
- é proibido que quadrados de duas partes de navios distintos tenham um *canto* em comum, ou seja, que quadrados de duas partes de navios distintos compartilhem um *vértice*;
- para que um navio de um jogador seja destruído por *disparos* de seu oponente é necessário que o oponente acerte todas as partes do navio, o que pode exigir um número indeterminado de disparos.

O jogo consiste em *disparos* alternados entre os dois jogadores, sendo que cada disparo que um jogador faz em *direção* ao tabuleiro do seu oponente deve ser feito tendo como *alvo* um único quadrado daquele tabuleiro. Para fazer um *disparo*, um jogador informa ao outro a linha L ($1 \leq L \leq n$) e a coluna C ($1 \leq C \leq m$) do quadrado alvo de seu disparo. Considere que os jogadores não se esquecem de seus disparos anteriores e, por isso, nunca *atiram* no mesmo lugar mais de uma vez.

Sua tarefa é escrever um programa em \mathbb{C} que *simule* uma partida deste jogo a partir da configuração do tabuleiro e de uma sequência de disparos feitos por um dos jogadores, determinando o número de navios do outro jogador que foram destruídos pela sequência de disparos.

Entrada

A primeira linha da entrada contém dois números inteiros n e m ($1 \leq n, m \leq 100$) representando, respectivamente, o número de linhas e de colunas do tabuleiro.

As n linhas seguintes correspondem ao tabuleiro do jogo. Cada uma dessas linhas contém m caracteres, sendo que cada caractere indica o conteúdo da posição correspondente no tabuleiro. Se esse caractere for 'a' (letra 'a', minúscula), essa posição contém água; se o caractere for '#' (hashtag), essa posição contém uma parte de um navio.

A próxima linha contém um número k ($1 \leq k \leq n \times m$) que representa o número de disparos feitos pelo jogador em direção ao tabuleiro de seu oponente.

As próximas k linhas indicam os *disparos* feitos pelo jogador, sendo que cada linha contém dois inteiros L e C , indicando a linha e a coluna do disparo feito, lembrando que $1 \leq L \leq n$ e $1 \leq C \leq m$.

Saída

Seu programa deve imprimir uma única linha contendo um único número natural: o número de navios destruídos do jogador oponente ao que realizou os disparos.

Exemplos

Entrada	Saída
5 5 aa#a# #aaaa aaa#a #aaaa aaa#a 5 1 3 1 4 1 5 2 1 3 4	4

Entrada	Saída
5 5 aa### aaaaa #### aaaaa #a##a 5 5 1 5 2 1 3 1 4 1 5	2

Entrada	Saída
<pre> 7 7 a#aaaa# ###-a# # a#aaaa# aaaa#a# a#aa#a# a###a# aaaaaaa 8 1 1 1 2 2 1 2 2 2 3 3 2 5 2 6 2 </pre>	1

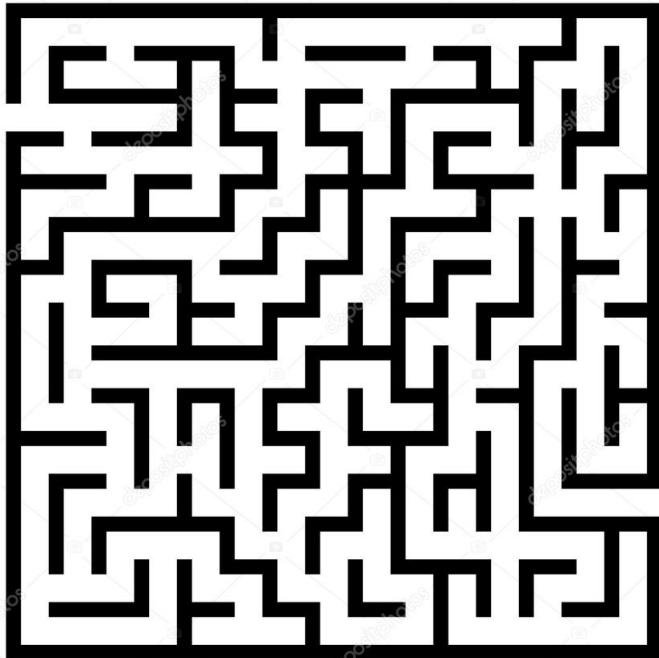
Observações

Se você domina alguma linguagem de programação que permita o desenvolvimento de programas que apresentem uma GUI (*Graphic User Interface*) para seu usuário, que tal pensar em desenvolver este *joguinho* a utilizando?

12 Labirinto – 1



(++++)



Considere que você está jogando um intrigante “*Jogo de Labirinto*”.

O labirinto é, em verdade, uma matriz $m \times n$, onde casa dessa matriz possui uma coordenada (x,y) da próxima casa onde você deverá ir, e a saída do labirinto sempre será a posição $(0,0)$.
Por exemplo, observando a figura abaixo temos que o você está na casa vermelha, que é a posição $(0,1)$ e dela você irá para a posição amarela $(1,2)$ e da posição amarela você irá para a posição verde $(0,0)$ – indicando que você conseguiu *sair* do labirinto e, portanto, venceu o jogo. Parabéns!

	0	1	2
0	0 , 0	1 , 2	1 , 1
1	0 , 2	2 , 2	0 , 0
2	2 , 2	0 , 0	0 , 2

Noutro exemplo, ao invés de iniciar na posição $(0,1)$, você iniciaria na posição $(1,0)$. Neste caso, você sairia da posição vermelha $(1,0)$ e iria para a posição azul $(0,2)$. De lá, você iria para a posição amarela $(1,1)$ e então iria para a posição verde $(2,2)$. Da posição verde, você voltaria para a posição azul $(0,2)$, o que caracteriza que você entrou em “*looping*”. Por isso, partindo da posição $(1,0)$ não é possível chegar à saída do labirinto, ou seja, é impossível ganhar o jogo a partir dela. Lamento!

	0	1	2
0	0 , 0	1 , 2	1 , 1
1	0 , 2	2 , 2	0 , 0
2	2 , 2	0 , 0	0 , 2

Você deverá escrever um programa C para *similar* este jogo de acordo com as especificações a seguir.

Entrada

A primeira linha da entrada contém as dimensões m e n da matriz, sendo o número de linhas e de colunas, respectivamente. Sabe-se que $m, n \in \mathbb{N}^*$ e que $1 \leq m, n \leq 100$.

As m linhas seguintes, contém, cada uma, os n pares de coordenadas de cada célula da matriz, com todos os números sendo separados por um único espaço em branco em relação seu anterior e posterior. Obviamente, o primeiro número da linha não tem anterior e o último número não tem posterior.

Por fim, a última linha contém as coordenadas da posição inicial, (x, y) , a partir de onde o jogo começará, representada por meio dos números x e y , separados por um único espaço em branco, e na ordem especificada: x seguido de y .

Saída

A palavra VENCE (em letras maiúsculas), se for possível ganhar o jogo, ou seja, sair o labirinto a partir de uma certa posição (x, y) inicial.

A palavra PRESO (em letras maiúsculas), caso seja impossível ganhar o jogo.

Exemplos

Entrada	Saída
<pre>3 3 0 0 1 2 1 1 0 2 2 2 0 0 2 2 0 0 0 2 0 1</pre>	VENCE

Entrada	Saída
<pre>3 3 0 0 1 2 1 1 0 2 2 2 0 0 2 2 0 0 0 2 1 0</pre>	PRESO

13 Labirinto – 2



(++++)



Considere que você **continua** jogando o “*Jogo de Labirinto*”. Entretanto, desta vez, na versão 2.0, ele ficou um pouco mais complicado, pois a posição inicial não será dada.

Você deve escrever um programa, em C, que dadas as dimensões m e n do labirinto, bem como os respectivos pares de cada casa, como no problema anterior, mas que seja capaz de calcular a “*quantidade de casas*” onde é possível chegar à saída, ou seja, quantas casas permitem que, iniciando-se a partir dela, seja possível ganhar o jogo.

Por exemplo, no tabuleiro a seguir, estão pintadas de vermelho todas as casas que, iniciando-se dela, atinge-se a saída. Neste caso, o programa deveria retornar 4.

	0	1	2
0	0 , 0	1 , 2	1 , 1
1	0 , 2	2 , 2	0 , 0
2	1 , 2	1 , 0	0 , 2

Entrada

A primeira linha da entrada contém as dimensões m e n da matriz, sendo o número de linhas e de colunas, respectivamente. Sabe-se que $m, n \in \mathbb{N}^*$ e que $1 \leq m, n \leq 100$.

As m linhas seguintes, contém, cada uma, os n pares de coordenadas de cada célula da matriz, com todos os números sendo separados por um único espaço em branco em relação seu anterior e posterior. Obviamente, o primeiro número da linha não tem anterior e o último número não tem posterior.

Saída

Uma única linha com a quantidade de casas a partir da qual é possível alcançar a saída – ganhar o jogo!

Exemplos

Entrada	Saída
3 3 0 0 1 2 1 1 0 2 2 2 0 0 1 2 1 0 0 2	4

Entrada	Saída
3 3 0 0 1 2 1 1 0 2 2 2 0 0 2 2 0 1 1 2	9

Entrada	Saída
5 5 0 0 2 0 3 0 1 0 1 1 0 4 3 1 0 2 2 4 2 1 4 4 2 3 1 3 4 2 4 1 0 0 1 4 3 4 2 2 4 3 0 1 4 0 3 2 0 3 1 2	13

14 Pegar e escapar



(++++)

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Considere que lhe é fornecido um “vetor” contendo n números naturais, com $1 \leq n \leq 30$.

Você deve escrever um programa, em \mathbb{C} , que seja capaz de escolher k números deste vetor de maneira tal que aplicando-se a operação lógica “xor” entre todos os k valores escolhidos, obtenha-se o *máximo* valor possível. Sabe-se que $1 \leq k \leq n$.

A operação “xor” deverá ser aplicada considerando-se a representação binária de cada um dos números.

Entrada

A primeira linha da entrada contém o número de casos de teste, t , a serem submetidos à avaliação. Sabe-se que $1 \leq t \leq 100$.

Cada caso de teste seguinte é formado por $(n + 1)$ linhas, onde:

- a primeira linha contém n e k daquele caso de teste, nesta ordem, e separados por um único espaço em branco;
- as n linhas seguintes contém, cada uma, o valor do respectivo elemento do vetor: 1º, 2º, 3º e assim sucessivamente. Sabe-se que o valor de cada elemento está entre 1 e 10000, inclusive extremos.

Saída

Imprima, para cada um dos t casos de teste, uma linha contendo o valor *máximo* obtido para a aplicação da operação “xor” dentre k elementos escolhidos naquele caso de teste.

Exemplo

Entrada	Saída
2 5 3 1 2 3 4 5 5 3 3 4 5 7 4	7 7

Entrada	Saída
2 10 2 10000 10000 10000 10000 10000 10000 10000 10000 10000 10000 10000 10000 10000 10 2 0 0 0 0 1 1 1 1 1	0 1

15 Altas aventuras



(+++++)



Incentivado por um filme de animação¹, vovô Gepeto resolveu realizar seu sonho de criança: fazer sua pequena casa voar amarrada a balões cheios de gás hélio. Comprou alguns balões coloridos, de boa qualidade, para fazer alguns testes e começou a planejar a sua grande aventura.

A primeira tarefa é determinar qual a quantidade máxima de gás hélio (He) que pode ser injetada em cada balão de maneira tal que ele não *estoure*. Para isto suponha que os valores possíveis de quantidade de gás hélio em cada balão variem, de maneira discreta, entre os valores 1 e n , sendo 1 a mínima quantidade e n a máxima quantidade.

É claro que vovô Gepeto poderia testar “todas as possibilidades” de enchimento dos balões. Evidentemente estetipo de solução ineficiente não é apropriada, ainda mais considerando que vovô Gepeto comprou apenas k balões para os seus testes, com $k \leq n$.

Por exemplo, suponha que $n = 5$ e que $k = 2$. Nesse caso, a melhor solução seria testar o primeiro balão com a quantidade de gás hélio igual a 3. Caso o balão estoure, vovô Gepeto só teria mais um balão, e então teria de testar os valores 1 e 2, no pior caso, somando ao todo três testes. Caso o balão não estoure com o primeiro valor (ou seja, o valor 3 neste caso), vovô poderia testar os valores 4 e depois 5 (ou 5 e depois 4), também somando três testes ao todo.

Observação: Considere que todos os balões tem igual resistência em relação à quantidade de gás hélio que suportam.

Tarefa

Dados a capacidade máxima da bomba disponível para enchimento dos balões ($n \in \mathbb{N}^*$) e o número de balões ($k \in \mathbb{N}^*$), indicar o “número mínimo de testes” que devem ser feitos, no pior caso, para determinar o ponto em que um balão estoura.

Entrada

A única linha da entrada contém dois números naturais, n e k , separados por um único espaço em branco ($1 < k \leq n \leq 1.0 \times 10^6$).

¹Up! Altas Aventuras, da Pixar Studios, 2009

Saída

Seu programa C deve imprimir uma única linha, contendo um número natural que representa o número mínimo de testes que devem ser feitos, no pior caso, para determinar o ponto em que o balão estoura.

Exemplos

Entrada	Saída
5 2	3

Entrada	Saída
20 2	6

Entrada	Saída
11 5	4

Universidade Federal de Goiás – UFG
Instituto de Informática – INF
Bacharelados (Núcleo Básico Comum)

Algoritmos e Estruturas de Dados 1 – 2020/2

Lista de Exercícios nº 03 – Tipo Abstrato de Dados (TAD)
Turmas: INF0286 – Prof. Wanderley de Souza Alencar)

Sumário

1 Conjuntos de Números Naturais	4
2 Manipulando Datas	7
3 Processamento de Textos	10
4 Mundo das Bactérias	12
5 Números Complexos	17

Observações:

- A resolução de cada um dos exercícios desta lista pressupõe a utilização do conceito de *Tipo Abstrato de Dados* (TAD) durante a implementação utilizando a linguagem de programação C ou C++;
- Detalhando: o único arquivo com a extensão .c deverá conter o que *deveria estar* em dois arquivos: o `tad.h` e o próprio `tad.c`;
- O uso do arquivo `tad.h` significa que a função `main()` elaborada como proposta de solução para o problema deve somente utilizar operações presentes neste arquivo.

```
#include <stdio.h>
#include <stdlib.h>

<TADs: conteúdo do(s) arquivo(s) .h>

<TADs: conteúdo do(s) arquivo(s) .c , sem #include “TAD.h”>

int main () {
    <seu código>
}
```

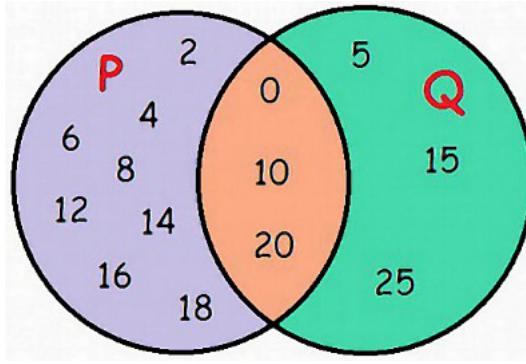
Veja o exemplo a seguir:

```
1 //=====
2 // Arquivo ponto.h
3 //=====
4 //
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <math.h>
8 #include <stdbool.h>
9
10 typedef struct ponto Ponto;
11 Ponto* ponto_cria(float x, float y, bool visibilidade);
12 void ponto_libera(Ponto* p);
13 void ponto_acessa(Ponto* p, float* x, float* y);
14 void ponto_atribui(Ponto* p, float x, float y);
15 float ponto_distancia(Ponto* p1, Ponto* p2);
16 void ponto_oculta(Ponto* p);
17 void ponto_mostra(Ponto* p);
18 void ponto_move(Ponto* p, float x, float y);
19 //
20 //=====
21 // Arquivo ponto.c
22 //=====
23 //
24 struct ponto {
25     float x;
26     float y;
27     bool visibilidade;
28 };
29 //
30 // Cria um ponto
31 //
32 Ponto* ponto_cria (float x, float y, bool visibilidade) {
33     Ponto* p = (Ponto*) malloc(sizeof(Ponto));
34     if(p != NULL) {
35         p->x = x;
36         p->y = y;
37         p->visibilidade = visibilidade;
38     }
39
40     return (p);
41 }
42 //
43 // Libera (desaloca) um ponto...
44 //
45 void ponto_libera (Ponto* p) {
46     if (p != NULL) {
47         free(p);
48     }
49 }
50 //
51 // Acessa um ponto, coletando suas coordenadas
52 //
53 void ponto_acessa (Ponto* p, float* x, float* y) {
54     if (p != NULL) {
55         *x=p->x;
56         *y=p->y;
57     }
58 }
59 //
60 // Atribui coordenadas a um ponto, modificando-o
61 //
62 void ponto_atribui (Ponto* p, float x, float y) {
63     if (p != NULL) {
64         p->x=x;
65         p->y=y;
66     }
67 }
68 //
69 // Retorna a distancia entre dois pontos
70 //
71 float ponto_distancia (Ponto* p1, Ponto* p2) {
72     float dx = p1->x - p2->x;
73     float dy = p1->y - p2->y;
74     return (sqrt(dx*dx+dy*dy));
75 }
```

```

76
77 // 
78 // Oculta (torna invisivel) o ponto
79 //
80 void ponto_oculta(Ponto* p) {
81
82     p->visibilidade = false;
83 }
84 //
85 // Mostra (torna visivel) o ponto
86 //
87 void ponto_mostra(Ponto* p) {
88
89     p->visibilidade = true;
90 }
91
92 void ponto_move(Ponto * p, float x, float y) {
93     //
94     // Codigo para movimentacao do ponto
95     //
96 }
97 //
98 // Corpo principal
99 //
100
101 int main(){
102     float xp,yp,xq,yq,d;
103     Ponto *p,*q;
104
105     printf("digite as coordenadas x e y para o ponto 1: ");
106     scanf("%f %f",&xp,&yp);
107     printf("digite as coordenadas x e y para o ponto 2: ");
108     scanf("%f %f",&xq,&yq);
109     p = ponto_cria(xp,yp,true);
110     q = ponto_cria(xq,yq,true);
111     d = ponto_distancia(p,q);
112     ponto_acessa(p,&xp,&yp); ponto_acessa(q,&xq,&yq);
113     printf("Distancia entre os pontos (%.2f,.2f) e (%.2f,.2f) = %.5f\n",xp,yp,xq,yq,d);
114     ponto_libera(p); ponto_libera(q);
115     return (0);
116 }
```

./programas/pontoCompleto.c



1 Conjuntos de Números Naturais



(++) A linguagem C não possui um *tipo de dado* que seja capaz de representar a ideia de *conjunto finito* conforme a acepção Matemática do termo, ou seja, “*Uma coleção finita de elementos (entes ou componentes), na qual a ordem e a repetição destes elementos é irrelevante e, por isso, desconsiderada.*”. Escreva, em C, um programa que seja capaz de representar um *conjunto de números naturais* por meio do uso do conceito de Tipo Abstrato de Dado (TAD).

O programa deve implementar, no mínimo, as seguintes operações fundamentais:

1. criar um conjunto C , inicialmente *vazio*:

```
int criaConjunto(C);
retornando SUCESSO ou FALHA.
```

A falha ocorre se não for possível, por alguma ocorrência, criar o conjunto C .

2. verificar se o conjunto C é vazio:

```
int conjuntoVazio(C);
retornando TRUE ou FALSE.
```

3. incluir o elemento x no conjunto C :

```
int insereElementoConjunto(x, C);
retornando SUCESSO ou FALHA.
```

A falha acontece quando o elemento x já está presente no conjunto C ou, por algum outro motivo, a inserção não pode ser realizada.

4. excluir o elemento x do conjunto C :

```
int excluiElementoConjunto(x, C);
retornando SUCESSO ou FALHA.
```

A falha acontece quando o elemento x não está presente no conjunto C ou, por algum outro motivo, a remoção não pode ser realizada.

5. calcular a cardinalidade do conjunto C :

```
int tamanhoConjunto(C);
retornando a quantidade de elementos em  $C$ . O valor 0 (zero) indica que o conjunto está vazio.
```

6. determinar a quantidade de elementos do conjunto C que são maiores que x :

```
int maior(x, C);
O valor 0 (zero) indica que todos os elementos de  $C$  são maiores que  $x$ .
```

7. determinar a quantidade de elementos do conjunto C que são menores que x :

```
int menor(x, C);
O valor 0 (zero) indica que todos os elementos de  $C$  são menores que  $x$ .
```

8. verificar se o elemento x pertence ao conjunto C :
`int pertenceConjunto(x, C);`
 retornando TRUE ou FALSE.
9. comparar se dois conjuntos, C_1 e C_2 são idênticos:
`int conjuntosIdenticos(C1, C2);`
 retornando TRUE ou FALSE.
10. identificar se o conjunto C_1 é subconjunto do conjunto C_2 :
`int subconjunto(C1, C2);`
 retornando TRUE ou FALSE.
11. gerar o complemento do conjunto C_1 em relação ao conjunto C_2 :
`Conjunto complemento(C1, C2);`
 retornando um *conjunto* que contém os elementos de C_2 que não pertencem a C_1 .
 Se todos os elementos de C_2 estão em C_1 , então deve retornar um conjunto vazio.
12. gerar a união do conjunto C_1 com o conjunto C_2 :
`Conjunto uniao(C1, C2);`
 retornando um *conjunto* que contém elementos que estão em C_1 ou em C_2 .
13. gerar a intersecção do conjunto C_1 com o conjunto C_2 :
`Conjunto interseccao(C1, C2);`
 retornando um *conjunto* que contém elementos que estão em C_1 e, simultaneamente, em C_2 .
 Se não houver elementos comuns deverá retornar um conjunto vazio.
14. gerar a diferença entre o conjunto C_1 e o conjunto C_2 :
`Conjunto diferenca(C1, C2);`
 retornando um *conjunto* que contém elementos de C_1 que não pertencem a C_2 .
 Se todos os elementos de C_1 estão em C_2 deve retornar um conjunto vazio.
15. gerar o conjunto das partes do conjunto C :
`Conjunto conjuntoPartes(C);`
16. mostrar os elementos presentes no conjunto C :
`void mostraConjunto(C, ordem);`
 Mostrar, no dispositivo de saída, os elementos de C .
 Se *ordem* for igual a CRESCENTE, os elementos de C devem ser mostrados em ordem crescente. Se *ordem* for igual a DECRESCENTE, os elementos de C devem ser mostrados em ordem decrescente.
Observação: Como o dispositivo típico de saída é o monitor de vídeo, o(a) programador(a) tem liberdade para definir como os elementos serão dispostos nele. Por exemplo: dez ou vinte elementos por linha. Noutro exemplo: o programa definirá quantos elementos mostrar, por linha, de acordo com o número de elementos existentes no conjunto a ser apresentado.
17. copiar o conjunto C_1 para o conjunto C_2 :
`int copiarConjunto(C1, C2);`
 retornando SUCESSO ou FALHA.
 A falha acontece quando, por algum motivo, não é possível copiar os elementos do conjunto C_1 para o conjunto C_2 .
18. destruir o conjunto C :
`int destroiConjunto(C);`

retornando SUCESSO ou FALHA.

A falha acontece quando, por algum motivo, não é possível eliminar o conjunto C da memória.

Observações: Considere que:

- SUCESSO = 1; FALHA = 0;
- TRUE = 1; FALSE = 0;
- CRESCENTE = 1; DECRESCENTE = 0;
- qualquer conjunto poderá ter no máximo 1.000.000 (um milhão) de elementos, ou seja, esta é a *cardinalidade máxima* de um conjunto. Se qualquer operação resultar num conjunto com cardinalidade maior, então a função correspondente deverá retornar um *conjunto vazio* (se ela retorna um conjunto) ou FALHA (se ela retorna SUCESSO ou FALHA);
- a biblioteca `limits.h` da linguagem C contém duas constantes para denotar quais são o *menor* e o *maior* `long int` que pode ser utilizado no ambiente computacional em que o programa está sendo elaborado. São elas: `LONG_MIN` e `LONG_MAX`. Elas deverão ser, respectivamente, o menor e o maior número que podem ser armazenados num conjunto qualquer do programa;
- os nomes das funções anteriormente apresentados no texto devem ser obedecidos, ou seja, o código-fonte C elaborado deverá obrigatoriamente utilizá-los. É claro que outras funções acessórias podem ser criadas livremente pelo(a) programador(a).



2 Manipulando Datas



(++)

É inquestionário que a capacidade de *manipular datas* é de extrema importância em muitas aplicações práticas na área de processamento de dados. Infelizmente nem sempre há, numa determinada linguagem de programação que se está utilizando para o desenvolvimento de aplicações, uma *biblioteca* com variadas funções para realizar a manipulação de datas.

Considere que você está participando do desenvolvimento de uma *biblioteca* para esta finalidade, sendo que ela deverá ser integralmente escrita em C e conter pelo menos as seguintes funções, expressas por seus cabeçalhos: `data.h`.

1. `Data * criaData (unsigned int dia, unsigned int mes, unsigned int ano);`
Cria, de maneira dinâmica, uma *data* a partir dos valores para dia, mês e ano fornecidos.

2. `Data * copiaData (Data d);`
Cria uma *cópia* da data *d*, retornando-a.

3. `void liberaData (Data * d);`
Destroi a data indicada por *d*.

4. `Data * somaDiasData (Data d, unsigned int dias);`
Retorna uma data que é *dias* dias posteriores à data *d*.
Por exemplo, fornecendo a data *d* = 16/03/2020 e *dias* = 5, retornará a data 21/03/2020.

5. `Data * subtrairDiasData (Data d, unsigned int dias);`
Retorna uma data que é *dias* dias anteriores à data *d*.
Por exemplo, fornecendo a data *d* = 16/03/2020 e *dias* = 15, retornará a data 01/04/2020.

6. `void atribuirData (Data * d, unsigned int dia, unsigned int mes, unsigned int ano);`
Atribui, à data *d*, a data dia/mes/ano especificada.
Se não for possível, então faz com que *d* seja alterada para NULL.

7. `unsigned int obtemDiaData (Data d);`
Retorna a componente dia da data d.
8. `unsigned int obtemMesData (Data d);`
Retorna a componente mes da data d.
9. `unsigned int obtemAnoData (Data d);`
Retorna a componente ano da data d.
10. `unsigned int bissextoData (Data d);`
Retorna TRUE se a data pertence a um ano bissexto. Do contrário, retorna FALSE.
11. `int comparaData (Data d1, Data d2);`
Retorna MENOR se $d1 < d2$, retorna IGUAL se $d1 = d2$ ou retorna MAIOR, se $d1 > d2$.
12. `unsigned int numeroDiasDatas (Data d1, Data d2);`
Retorna o número de dias que existe entre as datas d1 e d2.
Se $d1 = d2$, então o número de dias é igual a 0 (zero). Do contrário, será um número estritamente positivo.
13. `unsigned int numeroMesesDatas (Data d1, Data d2);`
Se d1 e d2 estão no mesmo mês/ano, então o número de meses é igual a 0 (zero). Do contrário, será um número estritamente positivo.
14. `unsigned int numeroAnosDatas (Data d1, Data d2);`
Se d1 e d2 estão no mesmo ano, então o número de anos é igual a 0 (zero). Do contrário, será um número estritamente positivo.
15. `unsigned int obtemDiaSemanaData (Data d);`
Retorna o *dia da semana* correspondente à data d.
Considerando que DOMINGO = 1; SEGUNDA-FEIRA = 2; ... ; SÁBADO = 7.
16. `char * imprimeData (Data d, char * formato);`
Retorna uma *string* com a data “*formatada*” de acordo com o especificado em formato.
Se formato = “ddmmaaaa”, então a *string* retornada deverá apresentar os dois dígitos do dia, os dois dígitos do mês e os quatro dígitos do ano, nesta ordem, e separados por uma (/ – barra). Por exemplo: “12/11/2019”.
Se formato = “aaaammdd”, então a *string* retornada deverá apresentar os quatro dígitos do ano, os dois dígitos do mês e os dois dígitos do dia, nesta ordem, e separados por uma (/ – barra).
Por exemplo: “2019/11/12”.
De maneira análoga, são válidas as seguintes *strings* de formatação:
 - “aaaa”;
 - “mm”;
 - “dd”;
 - “ddmm”.

Entrada e Saídas

Não serão fornecidas entradas/saídas para testes, pois o(a) estudante deverá apenas submeter o código-fonte por ele(a) elaborado no *Sharif Judge System* do INF/UFG.

O programa elaborado deverá ter um *menu* que permita ao usuário selecionar cada uma das operações supramencionadas, executá-la e, em seguida, retornar ao *menu* para escolher uma nova opção.

Para *finalizar o programa* o usuário deverá fornecer um entrada especial. Por exemplo, o número 0 (zero) como opção no *menu*.

O(A) estudante terá liberdade para escolher como implementar a funcionalidade de *menu*.

Observações

1. Uma data é formada por seu dia, mês e ano;
2. Considere que as datas a serem aplicadas ao sistema serão, sempre, no intervalo de 01/01/1900 a 31/12/2200;
3. Fique atento a um importante evento que ocorreu no mês de outubro de 1582 envolvendo o calendário Gregoriano – pesquise sobre isto antes de implementar todas as funções. Em particular a função `obtemDiaSemanaData (Data d);`
4. TRUE = 1; FALSE = 0;
5. A função `comparaData (Data d1, Data d2)` deve retornar:

MENOR quando $d1 < d2$;

IGUAL quando $d1 = d2$;

MAIOR quando $d1 > d2$.

com **MENOR** = -1; **IGUAL** = 0 e **MAIOR** = 1.



3 Processamento de Textos



(+++)

Vamos, agora, elaborar um TAD que seja capaz, de maneira simples, representar um *texto* e disponibilizar diversas funções que sejam capazes de manipulá-lo.

Neste contexto, um *texto* é concebido como sendo “uma sequência de caracteres sem nenhuma formatação especial, ou seja, não existe **negrito**, *italico* ou qualquer outro atributo especial aplicado sobre os caracteres que formam o texto: há simplesmente o caractere. Entretanto, ele pode ser uma letra maiúscula ou minúscula, um dígito ou um símbolo especial.

Considere que as seguintes operações estão previstas para estarem presentes no TAD `Texto.h`:

1. `Texto * criarTexto (char * t);`
2. `void liberarTexto (Texto * t);`
3. `unsigned int tamanhoTexto (Texto * t);`
4. `char * obterTexto (Texto * t);`
5. `void mostrarTexto (Texto *t, unsigned int colunas).`
6. `Texto * copiarTexto (Texto * t);`
7. `void substituirTexto (Texto * t, char * alteracao);`
8. `Texto * concatenarTextos (Texto * t1, Texto * t2);`
9. `char * obterSubtexto (Texto * t, unsigned int inicio, unsigned int tama-`
10. `unsigned int encontrarSubtexto (Texto * t, char * subtexto, unsigned int ocorrencia);`
11. `int compararTextos (Texto * t1, Texto * t2).`

Observações

1. O procedimento `substituirTexto` substitui o texto presente em `t` pelo texto recebido em `alteracao`, mesmo que eles tenham tamanhos diferentes;
2. A função `obterTexto` deverá retornar uma cadeia de caracteres com o texto armazenado;
3. A função `compararTextos` (`Texto * t1, Texto * t2`) deve retornar:

MENOR quando $t1 < t2$;

IGUAL quando $t1 = t2$;

MAIOR quando $t1 > t2$.

com $\text{MENOR} = -1$; $\text{IGUAL} = 0$ e $\text{MAIOR} = 1$.

4. A função `obterSubtexto` deverá retornar uma cadeia de caracteres que se inicia na `inicio`-ésima posição do texto `t` e conter `tamanho` caracteres de extensão.
A primeira posição do texto `t` é a de número 1 (um).
Se, a partir da posição `inicio` não for possível obter os `tamanho` caracteres solicitados, a função deverá retornar uma cadeia que se inicia na posição `inicio` do texto `t` e conter até seu último caractere.
Se a posição `inicio` for inválida, ou seja, menor que 1 (um) ou maior que o tamanho do texto `t`, a função deve retornar uma cadeia *nula*.
5. A função `encontrarSubtexto` deve *procurar* pela ocorrência de número `ocorrência` (1^{a} , 2^{a} , 3^{a} , ..., n^{a}) do `subtexto` em `t`.
Se encontrar esta ocorrência, deverá retornar a posição do *primeiro caractere* dela em `t`. Do contrário, deverá retornar 0 (zero), pois o primeiro caractere de `t` é considerado como sendo o de número 1 (um);
6. A função `mostrarTexto` deve apresentar `t` no dispositivo de saída do computador (normalmente o monitor de vídeo), de tal maneira que a cada linha do dispositivo de saída sejam apresentados `colunas` caracteres. Por consequência, o texto `t` poderá ocupar uma ou mais linhas em função de seu tamanho total.



4 Mundo das Bactérias



(+++++)

Um estudante do curso de Bacharelado em Ciências Biológicas precisa, durante o seu Trabalho de Conclusão de Curso (TCC), fazer uma sequência de experimentos com uma “*cultura de bactérias*” para comprovar hipóteses a respeito da *movimentação* de cada uma destas bactérias nesta cultura.

O problema do estudante é que para fazer experimentos no “*mundo real*” das bactérias, consome-se muito tempo e, sabendo que você é estudante de um dos cursos de bacharelado do INF/UFG, pediu que você elaborasse um programa de computador que tornasse possível fazer uma *simulação* de culturas de bactérias, o que acelerará a obtenção dos resultados desejados.

Você gostou do desafio e passou algumas horas conversando com o estudante e, ao final, imaginou que uma “*cultura de bactérias*” pode ser representada, computacionalmente, por meio de uma matriz bidimensional W (de *world*) que contém um par de números naturais (x_i, y_i) , com n linhas e m colunas. Você considerou que $n, m \in \mathbb{N}$ e $1 \leq n, m \leq 1000$, ou seja, que haverá no máximo 1.000.000 de bactérias numa cultura. Cada *bactéria* é representada por um par (x_i, y_i) , onde:

x_i é a identificação daquela *bactéria*, ou seja, o número associado a ela – que é chamada de *ordem* da bactéria. Sabe-se que $1 \leq x_i \leq (n \cdot m)$;

y_i é a *força* da bactéria em relação às demais bactérias de sua cultura ou de outras culturas.

A *força* é expressa por um número que varia de 1 a 100, inclusive extremos, sendo que a força máxima é 100 (cem) e a mínima é 1 (um).

z_i é a *expectativa de vida* da bactéria, ou seja, o tempo, expresso em número de *épocas*, que a bactéria viverá se não houver uma intercorrência durante sua vida. Você apenas precisará utilizar este conceito se for implementar a porção “*opcional*” desta questão: veja a seção específica ao final deste texto.

Como você está, neste momento, estudando os TADs, se propôs elaborar o programa solicitado utilizando este conceito e a linguagem de programação C, para tornar a aventura mais “*hard*”.

Depois de alguns rascunhos, você chegou à conclusão de que as seguintes operações devem ser disponibilizadas para o estudante de Ciências Biológicas por seu programa:

```
[01] World * newWorld (unsigned int n, unsigned int m):
```

Cria uma nova cultura de bactérias, com n linhas e m colunas de tamanho.
A cultura deverá, após a operação, ficar vazia, ou seja, não conterá nenhuma bactéria.

[02] `World * cloneWorld (World * w):`

Faz a cópia da cultura de bactérias presente em w , gerando um *clone* dela, mesmo que esta esteja vazia.

[03] `void freeWorld (World * w):`

Se a cultura de bactérias w existe, a destrói. Do contrário ignora a solicitação.

[04] `unsigned int randomWorld (World * w, unsigned int n):`

Insere, em posições aleatórias, livres e distintas da cultura w , n bactérias.

A ordem a ser utilizada para as bactérias a serem adicionadas se inicia no número natural seguinte ao associado à bactéria que possuir o *maior ordem* na cultura w .

A *força* de cada bactéria deverá ser gerada aleatoriamente na faixa de valores permitida, o mesmo acontecendo com o tempo de *vida* de cada bactéria.

Se não for possível inserir as n bactérias em w , a função deverá retornar FALHA. O sucesso será indicado retornando SUCESSO.

Observação: Se w estiver inicialmente vazia, então a ordem das bactérias deve ser iniciada em 1 (um) e, por consequência, terminar em n .

[05] `unsigned int addBacterium (World * w,`

`unsigned int n, unsigned int f, unsigned int e):`

Insere, numa posição livre aleatoriamente escolhida da cultura w , a bactéria cuja ordem é dada pelo número n , cuja *força* é dada por f e cuja *expectativa de vida* seja expressa por e .

Se a bactéria de ordem n já está na cultura ou se a cultura não possui espaço para nenhuma bactéria adicional, a função deve retornar FALHA. Na hipótese de sucesso, a função deve retornar SUCESSO;

[06] `unsigned int addBacteriumXY (World * w, unsigned int n,`
`unsigned int x, unsigned int y, unsigned int f, unsigned int e):`

Insere, na linha x e coluna y da cultura w , a bactéria cuja ordem é dada pelo número n , cuja *força* é f e cuja *expectativa de vida* seja expressa por e .

Se a bactéria de ordem n já está na cultura ou se a cultura não possui espaço para nenhuma bactéria adicional ou se a posição (x, y) estiver ocupada ou, ainda, se a expectativa e for inválida, a função deve retornar FALHA. Na hipótese de sucesso, a função deve retornar SUCESSO.

[07] `unsigned int killBacterium (World * w, unsigned int n):`

Mata (destrói) a bactéria cuja ordem é dada pelo número n , independentemente de sua localização na cultura.

Se a bactéria de ordem n não estiver na cultura, a função deve retornar FALHA. Na hipótese de sucesso, a função deve retornar SUCESSO.

[08] `unsigned int killBacteriumXY (World * w, unsigned int x, unsigned int y):`

Mata (destrói) a bactéria que está na linha x e coluna y da cultura w .

Se não há bactéria na posição (x, y) , a função deve retornar FALHA. Na hipótese de sucesso, a função deve retornar SUCESSO.

[09] `World * jointWorlds (World * w1, World * w2):`

Realiza a *união* das culturas de bactérias $w1$ e $w2$, gerando uma nova cultura.

A operação de união deve ser consensual, ou seja, não pode haver *colisão* entre as posições ocupadas por

nenhuma das bactérias proveniente das culturas originais: é uma *união pacífica*.

Se houver colisão, a função deverá retornar `NUL` para indicar que a união não é possível. Do contrário, retorna a *nova cultura* gerada é retornada.

Observação: Lembre-se de que a cultura gerada deverá ter seus indivíduos renumerados, ou seja, a ordem dos elementos pertencentes à cultura gerada deverá ser alterada. Faça isto *renumerando* as bactérias provenientes da cultura `w2` de maneira a dar sequência à máxima ordem existente na cultura `w1`.

[10] `World * warWorlds (World * w1, World *w2)`:

Coloca as culturas de bactérias `w1` e `w2` em *guerra*, gerando uma nova cultura com as bactérias sobreviventes de acordo com as seguintes regras:

1. se houver duas bactérias que deveriam, na nova cultura, ocupar a mesma posição, a bactéria *mais forte* fagocita a bactéria *mais fraca* e ocupa aquele lugar na nova cultura. Além disso, sua *nova força* é dada pela soma da sua própria *força* e a da bactéria que acabou de fagocitar. Se houver *empate* entre as forças de ambas bactérias originais, escolha aquela proveniente da cultura `w1` para ir para a nova cultura, com *força dobrada*. A *expectativa de vida* da bactéria resultante é a maior expectativa de vida dentre as bactérias originais.
Se a *força* da bactéria vencedora superar a força máxima estabelecida, a ela deverá ser atribuída a *força máxima*.
2. se não houver disputa, a bactéria deverá permanecer, na nova cultura, na mesma posição em que está na cultura de origem, com sua mesma *força* e *expectativa de vida* originais.

Observação: Lembre-se de que a cultura gerada deverá ter seus indivíduos renumerados, ou seja, a ordem dos elementos pertencentes à cultura `w` deverá ser alterada. Faça isto *renumerando* as bactérias provenientes da cultura `w2` de maneira a dar sequência à máxima ordem existente na cultura `w1`.

[11] `World * probabilisticWarWorlds (World * w1, World *w2, float p)`:

Coloca as culturas de bactérias `w1` e `w2` em guerra, gerando uma nova cultura com as bactérias sobreviventes de acordo com as seguintes regras:

1. se houver duas bactérias que deveriam, na nova cultura, ocupar a mesma posição, a bactéria *mais forte* fagocita a bactéria *mais fraca* de acordo com a probabilidade `p`, sabendo-se que $0 < p \leq 1$, e ocupa aquele lugar na nova cultura. Do contrário, a bactéria *mais fraca* é que fagocita a bactéria *mais forte* e ocupa aquele lugar na nova cultura.
Além disso, a *nova força* da bactéria vencedora é adicionada à da bactéria derrotada. Se a *força* da bactéria vencedora superar a força máxima estabelecida, a ela deverá ser atribuída a *força máxima*. A *expectativa de vida* da bactéria resultante é a maior expectativa de vida dentre as bactérias originais.
2. se não houver disputa, a bactéria deverá permanecer, na nova cultura, na mesma posição em que está na cultura de origem, com sua mesma *força* e *expectativa de vida* originais.

Observação: Lembre-se de que a cultura gerada deverá ter seus indivíduos renumerados, ou seja, a ordem dos elementos pertencentes à cultura `w` deverá ser alterada. Faça isto *renumerando* as bactérias provenientes da cultura `w2` de maneira a dar sequência à máxima ordem existente na cultura `w1`.

[12] `unsigned int sizeWorld (World * w)`:

Retorna o número de bactérias existentes na cultura `w`.

O valor 0 (zero) indicará que a cultura está vazia.

[13] `unsigned int forceWorld (World * w):`

Retorna a soma de todas as *forças* das bactérias existentes na cultura *w*.

O valor 0 (zero) indicará que a cultura está vazia.

[14] `unsigned int showWorld (World * w):`

Apresenta, no dispositivo de saída (normalmente o monitor de vídeo), uma “imagem” matricial da cultura *w* de acordo com seu tamanho (expresso por $n \cdot m$).

Observação: A proposta aqui é que você elabore, livremente, uma maneira de *mostrar* o que está presente na cultura *w* no dispositivo de saída.

Por exemplo, mesmo no modo texto, é possível construir um mapa como o representado abaixo, para uma cultura de tamanho $5 \cdot 5$:

	1	2	3	4	5
1	6,1	2,100	5,70		
2				4,32	3,3
3	7,4		8,3		
4		1,80			10,80
5				9,45	

O conteúdo da linha 1, coluna 1, indica que nesta célula está a bactéria de ordem 6 e cuja *força* é igual a 1. Na posição (2,4), portanto, está a bactéria de ordem 4 e de 32.

Apesar do exemplo, você estará livre para criar outras maneiras de mostrar a cultura de bactérias: solte sua imaginação para concebê-la, mesmo que isso faça com que você utilize a “*parte gráfica*” da linguagem.

Observação: Se a cultura *w* for *muito grande*, ou seja, a dimensão $n \cdot m$ não cabe inteiramente no dispositivo de saída, você poderá fazer com que o usuário escolha uma *porção* da cultura a ser visualizada. Isto pode ser feito fazendo com que ele selecione a porção das linhas e das colunas a serem mostradas.

Por exemplo, informando (1,10) e (6,14) significa que ele deseja visualizar as linhas de 1 a 10 e as colunas de 6 a 14. Evidentemente esta *porção* a ser visualizada não poderá superar a possibilidade de apresentação do dispositivo de saída. Se superar, avise-o da incorreção e peça para que ele selecione uma área menor.

IMPLEMENTAÇÃO OPCIONAL

A implementação das funções a seguir é **opcional**, pois elas são um *experimento* contínuo com o que acontecerá durante o processo de sucessivas interações entre duas culturas de bactérias.

Para aqueles(as) que gostam que “*quebra-cabeças*”, este é um bom exercício.

[15] `World * continuumWarWorlds (World * w1, World * w2):`

Coloca as culturas de bactérias *w1* e *w2* em *guerra contínua*, gerando uma sequência de novas culturas com as bactérias sobreviventes que, novamente, voltam à *guerra*.

Na *guerra contínua* é necessário entender o conceito de *época*: ele é um relógio que será iniciado em 0 (zero) marcando o número de batalhas já realizadas entre culturas de bactérias. Assim, este relógio contará continuamente: 0, 1, 2, 3, 4, ...

Todas as bactérias são consideradas “*nascidas*” na *época 0* (zero). Assim, se uma bactéria tiver *expectativa de vida* igual a 50, significa que ela poderá participar de até 50 batalhas, pois a cada batalha vencida sua *expectativa de vida* é diminuída de uma unidade. Quando a expectativa chegar a 0 (zero), aquela bactéria morre naturalmente, ou seja, sem participar de uma batalha.

É claro que uma bactéria poderá morrer antes do término de sua *expectativa de vida* por ter perdido uma batalha com outra bactéria (mais forte ou mais fraca se, por exemplo, se a disputa for a probabilística).

As regras para uma *batalha* desta guerra são as seguintes:

1. verifique quais as bactérias atingiram sua *expectativa de vida* e as elimine de ambas culturas – w_1 e w_2 – antes da batalha começar.
2. se houver duas bactérias que deveriam, na nova cultura, ocupar a mesma posição, a bactéria *mais forte* fagocita a bactéria *mais fraca* e ocupa aquele lugar na nova cultura. Além disso, sua *nova força* é adicionada à força da bactéria que acabou de fagocitar. Se houver *empate*, escolha a originária da cultura w_1 para ir para a nova cultura, com *força dobrada*. A *expectativa de vida* da bactéria resultante é a maior expectativa de vida dentre as bactérias originais.
Se a *força* da bactéria vencedora superar a força máxima estabelecida, a ela deverá ser atribuída a *força máxima*.
3. se não houver disputa, a bactéria deverá permanecer, na nova cultura, na mesma posição em que está na cultura de origem, com sua mesma *força* e *expectativa de vida* originais.

Ao terminar uma batalha, a nova cultura (ou seja, a gerada pela batalha das culturas w_1 e w_2). Vamos chamá-la de w) irá travar uma nova batalha com w_1 ou w_2 originais.

Quem irá para a batalha com w ?

Aquela que tiver a maior soma das forças de suas bactérias! (Lembre-se que a função:

`forceWorld (World * w)`

lhe fornecerá esta informação).

Assim, a cultura w tomará o lugar w_1 na nova batalha e a cultura *mais forte* entre w_1 e w_2 originais tomará o lugar de w_2 nesta nova batalha.

Esta guerra terá fim?

Esta é uma ótima questão para experimentação:

uma guerra terminará depois de uma sequência de DuracaoGuerra batalhas ou, se por algum motivo, haverá uma *estabilização* do processo de guerrilha que, por consequência, se tornará uma “Guerra Infinita”?

O valor de DuracaoGuerra será fixado por você durante a experimentação: 10, 20, 30, 50, ... 1000.

O que é uma *estabilização* da guerra?

Vamos considerar que a guerra ficou *estável* se após uma sequência de DuracaoEstabilizacao batalhas, não há mudança no valor retornado pela função `forceWorld` quando esta é aplicada na cultura vencedora da batalha.

Por exemplo: Depois de uma sequência de 10 batalhas (que seria o valor fixado para DuracaoEstabilizacao) a cultura vencedora está sempre com o mesmo valor de `forceWorld` obtido.

O valor DuracaoEstabilizacao terá que ser fixado para ser menor que o valor de DuracaoGuerra corrente.

Ao terminar a função deve retornar `NULL` se houve algum problema que a impediu de continuar até completar. Do contrário, deve retornar a cultura vencedora da guerra contínua.

Observação: Lembre-se de que a cultura gerada em cada batalha deverá ter seus indivíduos reenumerados, ou seja, a ordem dos elementos pertencentes à cultura w deverá ser alterada. Faça isto *renumerando* as bactérias provenientes da cultura w_2 de maneira a dar sequência à máxima ordem existente na cultura w_1 .

[16] `World * continuumProbabilisticWarWorlds (World * w1, World *w2, float p):`

É a versão probabilística da função:

`World * continuumWarWorlds (World * w1, World *w2),`

ou seja, ela utiliza as regras estabelecida por esta função para a guerra contínua, mas cada batalha entre w_1 e w_2 é realizada de acordo com as regras de probabilidade fixada em

`World * probabilisticWarWorlds (World * w1, World *w2, float p).`

Observações

- SUCESSO = 1; FALHA = 0;
- TRUE = 1; FALSE = 0;
- DuracaoGuerra varia entre 1 e 1000, inclusive extremos;
- DuracaoEstabilizacao varia entre 1 e o valor de DuracaoGuerra, inclusive extremos;
- a *expectativa de vida* de uma bactéria varia de 1 a 1000, inclusive extremos.

Complex Numbers

A Complex Number consist of a Real Part and an Imaginary Part

$a + bi$

Real Part
Imaginary Part

$i^2 = -1$
 $i = \sqrt{-1}$

5 Números Complexos



(+++)

Na ciência Matemática, um *número complexo* é um número expresso na forma $a + b \cdot i$, onde a e b são números reais ($a, b \in \mathbb{R}$) e i é um símbolo especial chamado de *unidade imaginária*, que satisfaz à equação $i^2 = -1$. Como nenhum número real é capaz de satisfazer a esta equação, o número que a satisfaz foi chamado, por René Descartes (1596–1650), de “*número imaginário*” e, atualmente, é representado pela letra i .

Para o número complexo $a + b \cdot i$, a é chamado de *parte real* do número, e b de *parte imaginária* do número. O conjunto dos números complexos é denotado por \mathbb{C} o que, em nosso caso, o torna idêntico à grafia utilizada para denotar a linguagem de programação C... mera coincidência.

Apesar de sua origem a nomenclatura de *imaginário*, os números complexos se mostram extremamente importantes para a Matemática e para diversas outras ciências, como a Física e a Engenharia, por exemplo. Eles nos auxiliam na resolução de muitos problemas do mundo natural que, sem eles, não teriam solução. Por exemplo: eles permitem encontrar uma solução para qualquer equação polinomial dada, mesmo para aquelas que não possuem solução real, comom, por exemplo, a equação $x^2 - x + 1 = 0$, cujas soluções são:

$$x_1 = \frac{1 + \sqrt{3} \cdot i}{2}$$

e

$$x_2 = \frac{1 - \sqrt{3} \cdot i}{2}$$

Outro exemplo é $(x + 1)^2 = -9$, cujas soluções são $x_1 = -1 + 3 \cdot i$ e $x_2 = -1 - 3 \cdot i$.

O que se deseja é que você, de maneira análoga ao que foi realizado na questão nº 1 desta lista, elabore, utilizando a linguagem C um programa que seja capaz de representar um *conjunto de números complexos* por meio do uso do conceito de Tipo Abstrato de Dado (TAD).

O programa deve implementar, no mínimo, as seguintes operações fundamentais:

1. criar um conjunto C , inicialmente *vazio*:

```
int criaConjunto(C);
retornando SUCESSO ou FALHA.
```

A falha ocorre se não for possível, por alguma ocorrência, criar o conjunto C .

2. verificar se o conjunto C é vazio:

```
int conjuntoVazio(C);
retornando TRUE ou FALSE.
```

3. incluir o elemento x no conjunto C :

```
int insereElementoConjunto(x, C);  
retornando SUCESSO ou FALHA.
```

A falha acontece quando o elemento x já está presente no conjunto C ou, por algum outro motivo, a inserção não pode ser realizada.

4. excluir o elemento x do conjunto C :

```
int excluiElementoConjunto(x, C);  
retornando SUCESSO ou FALHA.
```

A falha acontece quando o elemento x não está presente no conjunto C ou, por algum outro motivo, a remoção não pode ser realizada.

5. calcular a cardinalidade do conjunto C :

```
int tamanhoConjunto(C);  
retornando a quantidade de elementos em  $C$ . O valor 0 (zero) indica que o conjunto está vazio.
```

6. verificar se o elemento x pertence ao conjunto C :

```
int pertenceConjunto(x, C);  
retornando TRUE ou FALSE.
```

7. comparar se dois conjuntos, C_1 e C_2 são idênticos:

```
int conjuntosIdenticos(C1, C2);  
retornando TRUE ou FALSE.
```

8. identificar se o conjunto C_1 é subconjunto do conjunto C_2 :

```
int subconjunto(C1, C2);  
retornando TRUE ou FALSE.
```

9. gerar o complemento do conjunto C_1 em relação ao conjunto C_2 :

```
Conjunto complemento(C1, C2);  
retornando um conjunto que contém os elementos de  $C_2$  que não pertencem a  $C_1$ .  
Se todos os elementos de  $C_2$  estão em  $C_1$ , então deve retornar um conjunto vazio.
```

10. gerar a união do conjunto C_1 com o conjunto C_2 :

```
Conjunto uniao(C1, C2);  
retornando um conjunto que contém elementos que estão em  $C_1$  ou em  $C_2$ .
```

11. gerar a intersecção do conjunto C_1 com o conjunto C_2 :

```
Conjunto interseccao(C1, C2);  
retornando um conjunto que contém elementos que estão em  $C_1$  e, simultaneamente, em  $C_2$ .  
Se não houver elementos comuns deverá retornar um conjunto vazio.
```

12. gerar a diferença entre o conjunto C_1 e o conjunto C_2 :

```
Conjunto diferenca(C1, C2);  
retornando um conjunto que contém elementos de  $C_1$  que não pertencem a  $C_2$ .  
Se todos os elementos de  $C_1$  estão em  $C_2$  deve retornar um conjunto vazio.
```

13. mostrar os elementos presentes no conjunto C :

```
void mostraConjunto(C, ordem);  
Mostrar, no dispositivo de saída, os elementos de  $C$ .
```

Se $ordem$ for igual a CRESCENTE, os elementos de C devem ser mostrados em ordem crescente, primeiramente de sua “*parte real*” e, havendo empate, utilizar a “*parte imaginária*”. Se $ordem$ for igual a DECRESCENTE, os elementos de C devem ser mostrados em ordem decrescente, primeiramente de sua “*parte real*” e, havendo empate, utilizar a “*parte imaginária*”.

Observação: Como o dispositivo típico de saída é o monitor de vídeo, o(a) programador(a) tem liberdade para definir como os elementos serão dispostos nele. Por exemplo: dez ou vinte elementos por linha. Noutro exemplo: o programa definirá quantos elementos mostrar, por linha, de acordo com o número de elementos existentes no conjunto a ser apresentado.

14. copiar o conjunto C_1 para o conjunto C_2 :

```
int copiarConjunto(C1, C2);  
retornando SUCESSO ou FALHA.
```

A falha acontece quando, por algum motivo, não é possível copiar os elementos do conjunto C_1 para o conjunto C_2 .

15. destruir o conjunto C :

```
int destroiConjunto(C);  
retornando SUCESSO ou FALHA.
```

A falha acontece quando, por algum motivo, não é possível eliminar o conjunto C da memória.

Observações: Considere que:

- SUCESSO = 1; FALHA = 0;
- TRUE = 1; FALSE = 0;
- CRESCENTE = 1; DECRESCENTE = 0;
- qualquer conjunto poderá ter no máximo 1.000.000 (um milhão) de elementos, ou seja, esta é a *cardinalidade máxima* de um conjunto. Se qualquer operação resultar num conjunto com cardinalidade maior, então a função correspondente deverá retornar um *conjunto vazio* (se ela retorna um conjunto) ou FALHA (se ela retorna SUCESSO ou FALHA);
- a biblioteca `limits.h` da linguagem C contém duas constantes para denotar quais são o *menor* e o *maior* `long int` que pode ser utilizado no ambiente computacional em que o programa está sendo elaborado. São elas: `LONG_MIN` e `LONG_MAX`. Elas deverão ser, respectivamente, o menor e o maior número que podem ser armazenados num conjunto qualquer do programa;
- os nomes das funções anteriormente apresentados no texto devem ser obedecidos, ou seja, o código-fonte C elaborado deverá obrigatoriamente utilizá-los. É claro que outras funções acessórias podem ser criadas livremente pelo(a) programador(a).